Microsoft SQL Server 2005

שאילתות T-SQL

תרגום: **איציק בן-גן**

עריכה ועיצוב: **שרה עמיהוד, מירי אלעני**

עיצוב: **גלית גרבר-קטן**

עיצוב עטיפה: **שרון רז**

לשם שטף הקריאה כתוב ספר זה בלשון זכר בלבד. ספר זה מיועד לגברים ונשים כאחד ואין בכוונתנו להפלות או לפגוע בציבור המשתמשים/ות.

- **טלפון: 09-9564716, 1-700-7000-44**
- **פקס: 09-9571582**
- **דואר אלקטרוני: info@hod-ami.co.il**
- **אתר באינטרנט: www.hod-ami.co.il**

Microsoft SQL Server 2005

שאילתות T-SQL

Itzik Ben-Gan

Lubor Kollar, Dejan Sarka

Microsoft

הוצאת הוד-עמי

Inside Microsoft SQL Server 2005:

T-SQL Querying (Solid Quality Learning)

By Itzik Ben-Gan, Lubor Kollar, Dejan Sarka

ISBN 978-0-7356-2313-2

הוצאת הוד-עמי בע"מ

ת.ד. 6108 הרצליה 46160

טלפון: 09-9564716 פקס: 09-9571582

www.hod-ami.co.il

info@hod-ami.co.il

מהדורה ראשונה 2006

HOD-AMI Ltd.

P.O.B. 6108, Herzliya

ISRAEL, 2006

מסת"ב ISBN 965-361-381-2

BP -ל

$$eПi + 1 = 0$$

תוכן עניינים מקוצר

תוכן העניינים

הקדמה...

כאשר חשבתי על כתיבת ההקדמה לכרך זה, המילה "מומחה" ("master") עלתה במוחי עוד טרם פתחתי את הטיוטה. כפועל – "להתמחות" משמעותו לרכוש ידע מעמיק במשהו או להפוך להיות מיומן בשימושיו. כשם–עצם – "מומחה" היא מילה עשירה עם פרשנויות עדינות רבות. בהקשר של ספר זה עולות שתיים. מומחה כמישהו בעל כישורים מושלמים במיזם כלשהו ומומחה כמישהו מוסמך ללמד חניכים. כאשר פתחתי טיוטה מוקדמת הופתעתי לגלות, במעין סינכרוניות, את הפרק הראשון שפותח בדיון על "התמחות" בעקרונות היסוד של SQL.

המאפיין המדהים של שפת ה–SQL הוא הנגישות. תוכל ללמד אדם את הרעיונות הבסיסיים בתוך דקות ולהביא אותו מהר מאוד למצב בו הוא יכול לחקור את מסד הנתונים לבדו. SQL ניתן ללמוד בקלות אך לא בקלות מתמחים בה. התמחות בעושר של SQL דורשת זמן, אך הגמול רב. קיימות דרכים רבות לבצע דבר כלשהו בשפת SQL. במקרים רבים, כאשר אנשים מתחילים ללמוד SQL, הם מעצבים פתרונות לשאילתות שלהם בדרכים כואבות, עקומות ובלתי יעילות. מומחי SQL, מצד שני, לעיתים קרובות יוצרים פתרונות פשוטים ואלגנטיים שמהנה להסתכל עליהם, להעריך אותם וללמוד מהם. ספר זה מלא בדוגמאות לפתרונות פשוטים ואלגנטיים אלו, שיסייעו לך לשלוט ב–SQL.

בחיים אנו פוגשים בשני סוגים של מומחים; אלו החולקים את מומחיותם אך ורק דרך העבודה שלהם; ואלו הלוקחים חניכים לקדם את האמנות שלהם – להעביר אותה הלאה ולחלוק אותה עם אחרים. לקוראים של ספר זה יש את ההזדמנות להפוך לחניכים של SQL מפי מומחים של השפה ושל היישום הייחודי שלה ב– Microsoft SQL Server. איציק בן–גן, הסופר הראשי של ספר זה, הוא מומחה מוכר בתחום ה–SQL. איציק הוא אחד מהאנשים המגדירים מחדש את המדיום. לעיתים קרובות, כאשר אני מתבונן בפתרון שאיציק עיצב, אני נתקף בתחושה הזו של "לא ידעתי שאפשר לעשות את זה!". לובור קולר (Lubor Kollar), מנהל צוות בצוות פיתוח המוצר של Microsoft SQL Server, הוא מומחה בפתרון בעיות של לקוחות. לעיתים אני הלקוח. פעמים רבות נכנסתי למשרד של לובור בשאלה על דרך מסוימת לפתרון בעיה כלשהי ב–SQL, ובתוך דקות, לובור שלח לי פתרון שבסופו של דבר מצא את דרכו למחשב הנייד שלי לשימוש עתידי. דיין סרקה (Dejan Sarka), SQL Server MVP ידוע, הוא בעל ידע נרחב ביותר בתחום הרלציוני הטהור, אך הוא גם מומחה BI ובשילוב XML ו–CLR ב– SQL Server 2005. תרומתו העיקרית של דיין לספר זה היא בהדרכת הקוראים מתי רצוי לשלב XML או CLR בפתרונות שלהם.

הספר **שאילתות T-SQL** מתחיל היכן שצריך, בהפרדת המבנים הלוגיים של שפת SQL מהיישומים הפיסיים שלהם ב– Microsoft SQL Server. כמו במתמטיקה, נדרשת הבנה מעמיקה של המבנים הלוגיים עליהם SQL נוצרה, כדי לבנות בסיס יציב להמשך חקירה. פרקים 2 ו–3 נכנסים להיבטים של העיבוד הפיסי של שאילתות ומראים כיצד Microsoft SQL Server, המוצר, הופך שאילתה לוגית לתוכנית עבודה ניתנת לביצוע,

המחזירה את תוצאות השאילתה בצורה יעילה. רצוי להחזיק במושגים הללו של לוגי ופיסי בנפרד לחלוטין. עם זאת, ישנן מספר דרכים לתאר שאילתה בצורה לוגית, וחלק מהן מתורגמות ביתר קלות מאשר אחרות על ידי ה– SQL Server query optimizer לתוכניות יעילות. פרקים 4 עד 8 יוצרים את הבסיס לשליטה בשאילתות SQL. בפרקים אלה תלמד לבצע במספר שורות מלל של שאילתת SQL דברים שאנשים רבים (המשתמשים בשפת SQL אך עדיין לא שולטים בה) טורחים עליהם עם סמנים, טבלאות זמניות, או גרוע מכך, עם היגיון תכנות קשיח. כאן תלמד לשאול את השאלה כשאילתת SQL ותקבל את התשובה מ– SQL Server במקום לתכנת את התשובה בעצמך. תמצא שהפרקים האלה עשירים בדוגמאות רבות מהעולם האמיתי עם מספר יישומים חלופיים והסברים מתי להעדיף אחד על פני השני. לבסוף, פרק 9: "גרפים, עצים, היררכיות ושאילתות רקורסיביות" הוא ממלכתם של המומחים. אנשים רבים, שאינם מנוסים מספיק ב–SQL או במודל הרלציוני, מאמינים שמבני גרף אינם יכולים להיות מיוצגים בעולם הרלציוני. בעוד שברור כי דבר זה אינו נכון, ישנם כמה טיפולים בנושא בצורה מקיפה בטקסטים קיימים של SQL. פרק סיום זה מתאר כיצד לעצב גרפים ועצים דרך המהות שלהם – יחסים. הוא מציג מודל נתונים נפוץ לייצוג מבנים כגרפיים ואז מתאר מספר שיטות, כולל שימוש במאפיינים חדשים של SQL Server 2005, לנווט בין המבנים הללו.

הספר **שאילתות T-SQL** אינו רק ספר מתכונים. זה אינו המקום לחפש לגזור ולהדביק 20 שורות של קוד T-SQL לתוך התוכנית שלך. בעוד שהוא מכיל מתכונים ודוגמאות מפורשים, הספר **שאילתות T-SQL** הוא ספר על לימוד האמנות של SQL. על ידי קריאה בו תלמד לעשות דברים אותם חשבת לבלתי אפשריים לביצוע עם שפת SQL. כאשר תפגוש בעתיד אתגרים של SQL הדורשים שיטות שהוצגו בספר זה, תמצא את עצמך מושיט יד לספר זה שעל המדף שלך. בין אם אתה חדש בשפת SQL ו– Microsoft SQL Server, או שהינך עוסק בה, או שאתה כבר בדרכך להיות מומחה SQL, אני משוכנע שהספר הזה יסייע לך לקדם את האמנות שלך.

Microsoft SQL Server הוא כלי או פלטפורמה. כל אחד בצוות הפיתוח של המוצר מרגיש גאווה גדולה במסירת כלי נהדר. אך לבסוף – הערך האמיתי של SQL Server מובן רק כאשר הוא נמסר לידיו של מישהו שיוצר איתו משהו; בדיוק כפי שנגר משתמש בכלי ליצירת רהיט מובחר. כדי להיות נגר מומחה נדרש ידע של עקרונות היסוד של המדיום ושיטות לעבודה בו; חוש עיצוב מצוין; וידע מעמיק של כלי העבודה. אמנים מכירים את הכלים שלהם – כיצד ומתי להשתמש בהם; כמו גם כיצד לתחזק אותם. הספר **שאילתות T-SQL** יציג לך את עקרונות היסוד של שפת SQL; מקריאה בו, תבין בצורה ברורה כיצד לעצב פתרונות SQL אלגנטיים ממומחים של האומנות; לבסוף, תקבל הערכה להיבטים הייחודיים של Microsoft SQL Server ככלי לשימושך, כאשר אתה מתמחה בשפת SQL ליצירת פתרונות באמצעים פשוטים ואלגנטיים.

דיויד קמפבל
מנהל כללי, Microsoft SQL Server: אסטרטגיה, תשתית וארכיטקטורה.

מבוא

ספר זה וספר ההמשך – **Inside Microsoft SQL Server 2005: T-SQL Programming** (**MS SQL Server 2005: תכנות T-SQL**) – עוסקים בנושאים מתקדמים של ביצוע שאילתות, כוונון שאילתות ותכנות ב– SQL Server 2005. הם נכתבו עבור מפתחים ומנהלי מסדי נתונים הנדרשים לכתוב ולשפר קוד ב– SQL Server, הן בגרסת 2000 והן בגרסת 2005. בספר זה אתייחס לשני הספרים, ולשם הקיצור אתייחס לספרים בשמות **שאילתות T-SQL** ו**תכנות T-SQL**.

הספרים מתמקדים בבעיות מעשיות נפוצות, ודנים במספר גישות לפתרון כל אחת מהן. במהלך הספר תפגוש שיטות מלוטשות רבות שיעשירו את הכלים שברשותך ואת אוצר המילים התכנותי שלך, ויאפשרו לך לספק פתרונות יעילים בצורה טבעית.

הספרים חושפים את הכוח של שאילתות מבוססות–סטים, ומסבירים מדוע הן לרוב עדיפות על פני תכנות פרוצדורלי עם סמנים וכדומה. בו בזמן, הם ילמדו אותך כיצד לזהות את המקרים הבודדים בהם פתרונות מבוססי–סמן טובים יותר מאלו מבוססי–הסטים.

הספרים גם מציגים מבנים אחרים השנויים במחלוקת – כגון טבלאות זמניות, הפעלה דינמית, שילוב XML ו–NET. בכולם טמונים יכולות חזקות, אך בו בזמן גם סיכון רב. מבנים אלו דורשים בגרות תכנותית. ספרים אלו ילמדו אותך מתי יש להשתמש במבנים אלה, וכיצד לעשות זאת בצורה חכמה, בטוחה ויעילה.

ספר זה – **שאילתות T-SQL** – מתמקד בשאילתות מבוססות–סטים, ואני ממליץ לקרוא אותו ראשון. הספר השני – **תכנות T-SQL** (שבשלב זה לא ראה אור במהדורה עברית) – מתמקד בתכנות פרוצדורלי ומניח שקראת את הספר הראשון או שיש לך רקע מספק בתחום השאילתות.

הספר מתחיל בשלושה פרקים המניחים את הבסיס לעיבוד לוגי ופיסי של שאילתות, הנדרש לצורך הפקת המרב מיתר הפרקים.

הפרק הראשון עוסק בעיבוד לוגי של שאילתות. הוא מתאר בפירוט את השלבים הלוגיים המעורבים בעיבוד שאילתות, את ההיבטים הייחודיים של שאילתות SQL ואת הלך המחשבה המיוחד הדרוש כדי לתכנת בסביבה רלציונית (יחסית, Relational), מוכוונת סטים.

הפרק השני עוסק בעיבוד פיסי של שאילתות. הוא מתאר בפירוט את הדרך בה המנוע של SQL Server מעבד שאילתות, ומשווה מנגד עיבוד פיסי של שאילתות עם עיבוד לוגי שלהן. פרק זה נכתב על ידי לובור קולר (Lubor Kollar). לובור היה ראש צוות פיתוח בתקופת הפיתוח של SQL Server 2005 והצוות שלו היה אחראי לחלק "התחתון" של המנוע הרלציוני – מקומפילציה ואופטימיזציה של שאילתות לביצוע שאילתות, אחידות טרנזקציות, גיבוי/שחזור וזמינות גבוהה. המאפיינים הבולטים ב– SQL Server 2005 עליהם עבד הצוות שלו היו מחיצות של טבלאות ואינדקסים, שיקוף (Mirroring) של מסד נתונים, Snapshot של מסד נתונים, Snapshot Isolation, שאילתות רקורסיביות ושיפורים

נוספים ב– T-SQL, Database Tuning Advisor ויצירה ותחזוקה מקוונות של אינדקסים. בכל העולם סביר להניח שרק מספר קטן של אנשים מכיר את סוגיית האופטימיזציה של שאילתות טוב כמו לובור. אני רואה בכך פריבילגיה שאחד מהמעצבים של ה–optimizer מסביר אותו במילותיו הוא.

הפרק השלישי עוסק במתודולוגיה של כוונון שאילתות שפיתחנו בחברתנו (Solid Quality Learning) ויישמנו במערכות תפעוליות. הפרק עוסק גם בעבודה עם אינדקסים ובניתוח תוכניות עבודה. פרק זה מספק רקע חשוב הנדרש לפרקים שאחריו, העוסקים בעבודה עם אינדקסים וניתוח תוכניות עבודה. אלו היבטים חשובים של ביצוע וכוונון שאילתות.

הפרקים הבאים מתעמקים בהיבטים מתקדמים של ביצוע וכוונון שאילתות, בהם משתלבים הן ההיבטים הלוגיים והן ההיבטים הפיסיים של הקוד שלך. פרקים אלו כוללים: תת–שאילתות, ביטויי טבלה ופונקציות דירוג; Joins ופעולות סטים; סיכומי נתונים וסיבובם על ציר (כולל סעיף על סיכומים מוגדרי–משתמש ב–CLR, שנכתב על ידי דיין סרקה (Dejan Sarka)); TOP ו–APPLY; שינויי נתונים; והיררכיות ושאילתות רקורסיביות.

נספח א' עוסק בחידות היגיון. כאן יש לך הזדמנות לתרגל חידות היגיון כדי לשפר את הכישורים הלוגיים שלך. שאילתות SQL ביסודן עוסקות בלוגיקה. אני מוצא שחשוב לתרגל היגיון טהור כדי לשפר את יכולותיך לפתרון בעיות של שאילתות. אני גם מוצא בחידות אלו הנאה ואתגר, וניתן לתרגל אותן עם כל המשפחה. חידות אלו הן אוסף של חידות ההיגיון שפרסמתי בטור T-SQL שלי בירחון SQL Server Magazine. אני רוצה להודות ל– SQL Server Magazine שהרשו לי לחלוק את החידות הללו עם קוראי הספר.

הספר השני – **תכנות T-SQL** – מתמקד במבנים תכנותיים של T–SQL ומרחיב על שילוב של XML ו–NET. הנושאים אותם הוא כולל: בעיות הקשורות לטיפוסי נתונים, כולל טיפוסים מוגדרי–משתמש (UDT) ב–XML ו–CLR; טבלאות זמניות; סמנים; הפעלה דינמית; views; פונקציות מוגדרות–משתמש (UDF), כולל CLR UDFs; פרוצדורות מאוחסנות, כולל פרוצדורות CLR; triggers, כולל DDL ו– CLR triggers; טרנזקציות, כולל כיסוי של רמות הבידוד החדשות מבוססות–snapshot; טיפול בשגיאות ו– service broker.

הסעיפים בספר שעוסקים בשילוב של XML ו–NET. ("טיפוסים מוגדרי–לקוח", "פונקציות מוגדרות–לקוח", "פרוצדורות מאוחסנות" ו"Triggers") נכתבו על ידי דיין סרקה. דיין הוא מומחה SQL Server בעל ידע רב במודל הרלציוני (יחסי). הוא בעל השקפות מרתקות על הדרך בה מבנים חדשים אלו יכולים להתאים למודל הרלציוני כאשר משתמשים בהם בחוכמה. לדעתי חשוב שתחומים אלו של המוצר, השנויים במחלוקת, יוסברו על ידי מישהו עם תפיסה רחבה של המודל הרלציוני. כל דוגמאות הקוד של CLR מובאות הן ב–C# והן ב– Visual Basic .NET.

הפרק האחרון הכולל את Service Broker נכתב על ידי רוג'ר וולטר (Roger Wolter). רוג'ר הוא ראש צוות המפתחים של SQL Server שאחראי על ה– Service Broker. שוב, אין כמו מעצב הרכיב עצמו שיסביר עליו במילותיו הוא.

אחרון אך לא פחות חשוב, סטיב קאס (Steve Kass) הוא העורך הטכני של הספרים. סטיב הוא בחור חריף ביותר. הוא SQL Server MVP, והוא מלמד מתמטיקה באוניברסיטת דרו (Drew). יש לו ידע נרחב ב– SQL Server ולוגיקה, ותרומתו לספרים יקרת ערך.

ולך, הקורא של ספרים אלו, הייתי רוצה לומר שעבורי SQL היא מדע, לוגיקה ואומנות. בישלתי ספרים אלו זמן רב, ושפכתי לתוכם את כל הלהט שלי ושנים רבות של ניסיון. אני מקווה שתמצא את הספרים שימושיים ומעניינים, ושתמצא ב–SQL מקור השראה כפי שהיא לי. אם יש לך הערות או תיקונים שתרצה לחלוק, אשמח לשמוע. ניתן ליצור איתי קשר דרך **http://www.insidetsql.com**.

בהוקרה,

איציק בן–גן

תודות

מרבית הקוראים בדרך כלל מדלגים על פרק התודות, וסופרים רבים בדרך כלל כותבים אותו קצר מאוד. באמת שאיני רוצה לשפוט אף אחד; אני יכול רק לנחש שסופרים חושבים שחשיפת רגשותיהם עלולה להיראות קיטשית, או שהם נבוכים לעשות כך. ובכן, אנשים רבים תרמו את לבם ונשמתם לספרים אלו, ולא אכפת לי איך פרק זה עלול להיראות. אני רוצה שהם יוכרו בתרומתם!

הכרת תודה עמוקה ביותר שלי לכל אלו שלקחו חלק או תרמו בכל דרך לספרים. חלקם השקיעו שעות רבות במעורבות ישירה בפרויקט, ולחלקם הייתה השפעה עלי ועל עבודתי, שהשפיעה על הספרים באופן עקיף.

לסופרים האורחים לובור קולר, דיין סרקה ורוג'ר וולטר: תודה שלקחתם חלק בפרויקט זה והוספתם את התובנות יקרות הערך שלכם. זה היה כבוד ועונג לעבוד איתכם. לובור, עומק הידע והלהט שלך הם מקור השראה. דייקו, חברי הטוב, אני תמיד לומד ממך דברים חדשים. אני מוצא שההשקפות שלך על עולם מסדי הנתונים פשוט מרתקות. רוג'ר, אני מעריך מאוד את העובדה שהסכמת לתרום לספרים. Service Broker – התינוק שלך – מביא ממד חשוב ל– SQL Server שלא היה שם קודם לכן. אני בטוח שחברות ימצאו בו ערך רב, ואני להוט לראות כיצד טכנולוגיית התורים תיושם – יש לה פוטנציאל כה רב.

לסטיב קאס, העורך הטכני של הספרים: סטיב, אין לי מילים לתאר עד כמה אני מעריך את התרומה שלך. אתה פשוט גאוני ומדהים, ואדם יכול רק לקוות שיהיה לו מחצית מהתבונה וההיגיון בהם בורכת. תזכיר לי לעולם לא להגיע לקרב מוחות איתך. השקעת כל–כך הרבה זמן בפרויקט וסיפקת כל–כך הרבה הצעות בעלות תובנה עד כי אני מרגיש שלמעשה סייעת לכתוב את הספרים. אני מקווה שבמהדורות עתידיות תמלא תפקיד כתיבה רשמי.

לדיויד קמפבל (David Campbell) ולובור קולר שכתבו את ההקדמות: העבודה וההישגים שלכם הם אור מנחה לרבים מאיתנו. SQL Server גדל להיות מוצר בוגר ומרתק – כזה ששווה בהחלט להקדיש לו את הקריירות המקצועיות שלנו ולמקד את הלהט שלנו בו. תודה לשניכם על שהסכמתם לכתוב את ההקדמות. זהו כבוד אמיתי! לכל התורמים, אני מצפה לעבודה משותפת על פרויקטים בעתיד. אני בכל אופן, כבר התחלתי לבשל רעיונות למהדורות עתידיות של הספרים.

הרבה תודות לצוות בהוצאת הספרים של מיקרוסופט (Microsoft Press): בן ריאן (Ben Ryan), קריסטין האגסט (Kristine Haugset), רוג'ר לבלאנק (Roger LeBlanc), וכפי הנראה רבים אחרים שלקחו חלק בהכנת הספרים. בן, אני מצטער שהייתי כל–כך מעיק, ושרציתי להיות מעורב בכל פרט קטן שיכולתי. מושלמות וכנות הם שני הקווים שמנחים אותי, על אף שאפשר רק לשאוף לראשון. אני מאמין שעל ידי הליכה בדרך זו, התוצאה הסופית יכולה רק להשתפר, ובלי קשר, אני מאמין שאין דרך אחרת. תודה שהיית כה קשוב. קריסטין, את פשוט נהדרת! מסורה, מקצועית, אכפתית ומנווטת את הפרויקט בצורה כה אלגנטית. ברמה אישית יותר, אני מרגיש שהרווחתי חברה חדשה. רוג'ר, אני

לא מקנא בך על השעות הרבות מספור שהשקעת בעריכת הספרים. תודה שעזרת לשפר את איכותם. ואני בטוח שהיו רבים נוספים בהוצאה שעבדו שעות ארוכות מאחורי הקלעים כדי לאפשר לספרים לראות אור יום.

אני רוצה להודות לקיילן דלייני (Kalen Delaney). **ספרי Inside SQL Server** הקודמים של קיילן היו תנ"ך עבורי בכל הקשור ל– internals של SQL Server, ואני להוט לקרוא את הסקירה שלה על internals ב– SQL Server 2005 בכרכים החדשים. קיילן הייתה גם זו שביקשה ממני לראשונה לכתוב את הכרכים של T-SQL כעת כשהמוצר גדל כל–כך.

אנשים רבים סיפקו משוב יקר ערך ביותר על ידי ביקורת לא רשמית של הספרים. ביניהם חברים בצוות הפיתוח של SQL Server, מנטורים מ– Solid Quality Learning ו–MVPs. לא הייתם חייבים לעשות זאת, ועשיתם זאת בהתנדבות – נתונה לכם הכרת תודתי הכנה.

לצוות ב– SQL Server Magazine: עבורי אתם משפחה. אנחנו עובדים ביחד שנים וגדלנו והתפתחנו ביחד. אני בטוח שאיני היחיד שמאמין ש– SQL Server Magazine הוא המגזין הטוב בעולם למקצועני SQL Server. ספרים אלו בחלקם הגדול הם בזכות מה שספגתי ולמדתי בעבודה אתכם.

לחברי, שותפי ועמיתי ב– Solid Quality Learning: החברה הזו היא ללא ספק הדבר הטוב ביותר שקרה לי בקריירה המקצועית שלי, ומבחינות רבות בחיי האישיים. זה פשוט חלום שהתגשם. בהקשר לספרים, רבים מכם תרמו להם רבות דרך הביקורת שלכם ודרך העבודה שביצענו יחד, שכמובן משתקפת בספרים. אך הרבה מעבר לכך, אתם משפחה, חברים, ויותר מכל מה שאפשר לבקש. אני עדיין צריך לצבוט את עצמי כדי להיות בטוח שזו מציאות ולא חלום. ולפרנגדו: כולנו שותפים לדעה שאתה הסיבה לכך ש– Solid Quality Learning קמה לתחייה, ושבלעדיך היא לא הייתה קיימת. החזון שלך, הסבלנות, תשומת הלב, הלהט והלב הם השראה לכולנו. אין מילים שאוכל להגיד או לכתוב שיבטאו את הכרת התודה שיש בלבי לך ועד כמה אני מעריך את החברות בינינו.

אני מוצא שישנם שלושה מרכיבי מפתח המעצבים את הידע והיכולת של אדם: מורים, תלמידים ולהט. להט הוא הזרע שחייב לבוא מתוך האדם, ויש בי להט רב ל–SQL. בורכתי במורים נפלאים – לובור קולר וסנסאי יהודה פנטנוביץ'. לובור, הלהט והידע הנרחבים שלך של SQL והאופטימיזציה שלו, החיפוש שלך אחר ידע חדש, הצניעות שלך, הניצוץ בעיניך והחיבה שלך להיגיון, הם כולם מקור השראה בשבילי. אני אסיר תודה על שהכרנו, ואני מוקיר את החברות שלנו. סנסאי יהודה, על אף שייתכן שנראה לך כי לעולם שלך אין כל קשר ל–SQL, עבורי יש לו קשר חזק. העצה שלך, ההוראה, ההכוונה והחברות עזרה לי בעולם ה–SQL בדרכים רבות מכפי שתדע. אתה וסנסאי היגאונה (Higaonna) הנכם מקורות השראה אדירים בשבילי; אתם הוכחה חיה שאין דבר שהוא בלתי אפשרי – שאם נעבוד בשקידה ונשאף תמיד לשלמות, נוכל להשתפר ולהשיג דברים עצומים בחיים. ההתמקדות בפרטים קטנים, לעולם לא להיכנע, שליטה בהתרגשות על ידי המחשבה ש"זהו רק עוד יום במשרד", כנות, התמודדות עם הקטעים הרעים בחיים... אלו רק מספר דוגמאות של עצות שעזרו לי בדרכים רבות. כתיבה במשך שעות ארוכות

הייתה קשה ביותר, אך אני מייחס את העובדה שהצלחתי לעשות זאת בעיקר לך. תמיד חשבתי על הניסיון כאימון גוג'ו ארוך.

לסטודנטים שלי: למדתי, ואני ממשיך ללמוד המון דרך הוראה. למעשה, הוראה היא התשוקה האמיתית שלי, ואתם הסיבה העיקרית לכך שכתבתי ספרים אלו. בשנים האחרונות, טיילתי מסביב לעולם כדי ללמד. ביליתי זמן מועט מאוד בבית, והקרבתי הרבה לטובת התשוקה הזו. התקווה האמיתי שלי היא שלאחר שתקראו ספרים אלו, ניפגש בכיתת–הדוג'ו (חדר אימונים) המועדפת עלי ללימוד ותרגול של SQL.

להורי: החרטה היחידה שלי בכך שאני נוסע כל–כך הרבה, היא שזה בא על חשבון הזמן שלי אתכם. אני זוכר רגע סוריאליסטי שבאתם להרצאה שהעברתי על טבלאות מחולקות ואינדקסים ב– SQL Server 2005, כדי לראות אותי לאחר שלא התראינו שלושה חודשים. אני מתנצל ומקווה שאתם מבינים שאני צריך לרדוף אחר התשוקה שלי כדי להגיע להגשמה עצמית. אבא, תודה על כל העזרה במתמטיקה והיגיון. אמא, תפסיקי לצעוק על אבא כשהוא נותן לי חידה חדשה בשיחת טלפון טראנס–אטלאנטית; הוא לא יכול לעמוד בפיתוי וגם אני לא.

לילך, האהבה והעוגן שלי: הדבר היחידי ששומר על השפיות שלי כשאני רחוק מהבית הוא העובדה שאת איתי. אני חושב שאת היתר אני אגיד לך אישית; אנחנו לא רוצים להביך את עצמנו מול הקוראים.

אודות הספר

ספר זה מיועד לתוכניתני T-SQL ומנהלי מסדי נתונים מנוסים הנדרשים לכתוב שאילתות T-SQL. הספר עוסק בשאילתות T-SQL מתקדמות ומניח שיש לך כבר הבנה טובה של הרמה הבסיסית והבינונית של הנושאים (לפחות שנה של ניסיון בכתיבת קוד T-SQL) ושאתה מוכן להמשיך לרמה הבאה.

מבנה הספר

ספר זה הוא הראשון בשני כרכים, והוא מכסה שאילתות T-SQL מתקדמות. הכרך השני (שכרגע לא מתורגם לעברית) — Inside Microsoft SQL Server 2005: T-SQL Programming — מכסה תכנות מתקדם ב–T-SQL. הכרך השני מניח שקראת את הראשון או שיש לך רקע מקביל. לפרטים נוספים על המבנה של שני הכרכים, אנא פנה למבוא של ספר זה.

דרישות מערכת

כדי לבנות ולהריץ את דוגמאות הקוד בספר זה תידרש לרכיבי החומרה ולתוכנות הבאות:

- Microsoft Windows XP with Service Pack 2, Microsoft Windows Server 2003 with Service Pack 1, or Microsoft Windows 2000 with Service Pack 4
- Microsoft SQL Server 2005 Standard, Developer, or Enterprise Edition
- Microsoft Visual Studio 2005 Standard Edition or Microsoft Visual Studio 2005 Professional Edition
- 600-MHz Pentium or compatible processor (1-GHz Pentium recommended)
- 512 MB RAM (1 GB or more recommended)
- Video (800 by 600 or higher resolution) monitor with at least 256 colors (1024 by 768 High Color 16-bit recommended)
- CD–ROM or DVD–ROM drive
- Microsoft mouse or compatible pointing device

לפרטים נוספים על דרישות המערכת ועל התקנת SQL Server 2005, אנא פנה לפרק "Preparing to Install SQL Server 2005" ב–SQL Server Books Online (מותקן עם המוצר).

התקנת מסדי נתונים לדוגמה

ספר זה דורש ממך ליצור ולהשתמש במסדי נתונים, לדוגמה Northwind ו–pubs, אשר אינם מותקנים כברירת מחדל ב– SQL Server 2005. הקוד ליצירת מסדי נתונים אלו ניתן להורדה ממרכז ההורדות של מיקרוסופט בכתובת הבאה:

http://go.microsoft.com/fwlink/?LinkId=30196

לחלופין, תוכל להוריד את הקוד המייצר את מסדי נתונים אלו כחלק מדוגמאות הקוד עבור הספר. הוראות להורדה של דוגמאות הקוד עבור הספר יובאו מייד.

עדכונים

תוכל למצוא את העדכונים האחרונים של SQL Server בכתובת הבאה:

http://www.microsoft.com/sql/

תוכל למצוא מקורות הקשורים לספר בכתובת הבאה:

http://www.insidetsql.com/

דוגמאות קוד

כל דוגמאות הקוד הנידונות בספר זה ניתנות להורדה בכתובת הבאה:

http://www.insidetsql.com/

עברית

כן, אפשר ללמוד T-SQL בעזרת ספר הכתוב עברית. במהלך הקריאה והלימוד תראה שהדבר אפשרי.

מחבר הספר, איציק בן–גן, ישראלי יליד הארץ, המתגורר בה בדרך קבע, מעביר הרצאות וקורסים בכל העולם וגם בישראל, יודע כיצד לפנות לקהל מאזיניו דוברי העברית.

* המונחים **Attributes** ו–**Properties** תורגמו שניהם למילה **מאפיינים**.

תמיכה לספר זה

נעשו כל המאמצים כדי להבטיח את הדיוק של ספר זה ושל החומר המלווה אותו. עם זאת, שגיאות הן כנראה דבר בלתי נמנע. דף תיקונים ניתן למצוא בכתובת:

http://www.insidetsql.com/

התיקונים מתייחסים למהדורה האנגלית של הספר. תיקונים למהדורה העברית יצוינו בנפרד.

אם יש לך הערות, שאלות, או רעיונות בנוגע לספר זה, תוכל ליצור איתי קשר באתר הספר בכתובת לעיל.

1

עיבוד לוגי של שאילתות

מהתבוננות במומחים בתחומים שונים, ניתן להבחין כי יש נוהג המשותף לכולם – שליטה בעקרונות ובשיטות היסוד. בצורה זו או אחרת, כל המקצועניים מתמודדים עם פתרון בעיות. כל הפתרונות לבעיות, מורכבות ככל שיהיו, משלבים תערובת של שיטות מפתח. אם ברצונך להתמחות במקצוע, עליך לבנות את הידע שלך על יסודות מוצקים. השקע מאמצים רבים בשיפור שיטות העבודה שלך; שלוט בעקרונות היסוד, ותוכל לפתור כל בעיה.

ספר זה עוסק בשאילתות ב– **Transact-SQL (T-SQL)** – לימוד טכניקות מפתח ושימוש בהן כדי לפתור בעיות. אינני יכול לחשוב על דרך טובה יותר להתחיל את הספר מאשר בפרק על העקרונות של עיבוד לוגי של שאילתות. אני מוצא שפרק זה הוא החשוב ביותר בספר – לא רק מכיוון שהוא מכסה את היסודות של עיבוד שאילתות, אלא גם מכיוון שתכנות ב–SQL שונה מאוד, תפיסתית, מכל סוג תכנות אחר.

הניב של Microsoft SQL Server - Transact-SQL – תואם לתקן של ANSI. Microsoft SQL Server 2000 תואם לתקן ANSI SQL:1992 ברמת ה–Entry, ו– Microsoft SQL Server 2005 מיישם מספר מאפיינים חשובים של ANSI SQL:1999 ושל ANSI SQL:2003.

לאורך הספר אשתמש לעיתים במונח SQL ולעיתים במונח T-SQL. כאשר אדון בהיבטים של השפה אשר נובעים מ– ANSI SQL ורלוונטיים לרוב הניבים, אשתמש לרוב במונח SQL. כאשר אדון בהיבטים של השפה מתוך מחשבה על יישום של SQL Server, אשתמש לרוב במונח T-SQL. יש לציין כי השם הרשמי של השפה הוא Transact-SQL, על אף שלרוב היא נקראת T-SQL. מרבית התוכניתנים, ואני ביניהם, חשים נוח יותר לקרוא לשפה T-SQL, כך שעשיתי החלטה מודעת להשתמש במונח זה לאורך כל הספר.

המקור להגייה של SQL

מקצועני מסדי נתונים דוברי אנגלית רבים הוגים את המילה SQL כ–sequel, למרות שההגייה הנכונה של שם השפה היא S-Q-L ("אֶס קְיוּ אֶל").

ניתן לנסות להציע ניחושים מלומדים לסיבה להגייה השגויה. הניחוש שלי הוא שהסיבות הן גם היסטוריות וגם לשוניות.

באשר לסיבות היסטוריות, בשנות ה–70 של המאה העשרים, IBM פיתחה שפה שנקראה SEQUEL, ראשי התיבות של Structured English QUEry Language (שפת אנגלית מובנית לשאילתות). השפה עוצבה כדי לשלוט בנתונים שאוחסנו במערכת מסדי נתונים שנקראה מערכת R, אשר הייתה מבוססת על המודל של דר' אדגר פ. קוד למערכות ניהול מסדי נתונים רלציונים (RDBMS). מאוחר יותר, ראשי התיבות SEQUEL קוצרו ל–SQL בעקבות ויכוח על סימן מסחרי. ANSI אימצה את SQL כתקן בשנת 1986, ו–ISO עשה כמותה בשנת 1987. ANSI הכריזה שההגייה הרשמית של שם השפה היא "אֶס קְיוּ אֶל", אך נראה כי עובדה זו אינה ידועה לכל.

באשר לסיבות לשוניות, ההגייה sequel פשוט "זורמת" יותר, בעיקר לדוברי אנגלית. אני חייב לומר שאני משתמש בהגייה זו לעיתים קרובות בעצמי, בדיוק מאותה סיבה.

לעיתים ניתן לנחש באיזו הגייה אנשים משתמשים על ידי עיון בדברים שכתבו. מישהו שכותב: "an SQL Server" משתמש, סביר להניח, בצורת ההגייה הנכונה, בעוד מישהו שכותב "a SQL Server" משתמש, כפי הנראה, בצורת ההגייה השגויה.

מידע נוסף: אני ממליץ לקרוא על ההיסטוריה של SQL ועל ההגייה של שם השפה, נושא מרתק בעיני, בכתובת: http://www.wikimirror.com/SQL. סקירת ההיסטוריה של SQL באתר Wikimirror ובפרק זה מבוססים על מאמר מהאנציקלופדיה החופשית ויקפדיה.

ישנם היבטים ייחודיים רבים של תכנות SQL, כגון חשיבה בסטים, סדר העיבוד הלוגי של פסוקיות השאילתה ולוגיקת שלושת–הערכים. ניסיון לתכנת ב–SQL ללא הידע הזה הוא מרשם לקוד ארוך, בעל ביצועים גרועים וקשה לתחזוקה. המטרה של פרק זה היא לסייע לך להבין את שפת SQL כפי שהמעצבים שלה חזו אותה. עליך ליצור שורשים עמוקים עליהם ייבנה כל היתר. כאשר הדבר יהיה רלוונטי, אציין במפורש פסוקיות אשר הינן ספציפיות ל–T-SQL.

לאורך כל הספר אכסה בעיות מורכבות וטכניקות מתקדמות. אך בפרק זה, כפי שציינתי, אעסוק אך ורק בעקרונות היסוד של השאילתות. לאורך כל הספר אתמקד בביצועים. אך בפרק זה אעסוק אך ורק בהיבטים הלוגיים של עיבוד שאילתות. אבקש ממך לעשות מאמץ, כאשר אתה קורא פרק זה, ולא לחשוב בשלב זה על ביצועים. כמה מהשלבים של עיבוד לוגי של שאילתות שאציג בפרק זה עשויים להיראות מאוד לא יעילים. אך זכור שבפועל, העיבוד הפיסי של שאילתות עשוי להיות שונה מאוד מזה הלוגי.

הרכיב ב– SQL Server האחראי על הפקת תוכנית העבודה (execution plan) עבור שאילתה הוא ה– query optimizer. ה–optimizer קובע באיזה סדר לגשת לטבלאות, באילו שיטות גישה ואינדקסים להשתמש, איזה אלגוריתמים של join ליישם וכו'. ה–optimizer מייצר מספר תוכניות עבודה אפשריות ובוחר בזו בעלת העלויות הנמוכות ביותר. לשלבים בעיבוד הלוגי של שאילתה יש סדר מאוד מסוים. מצד שני, ה–optimizer יכול לעיתים קרובות לעשות קיצורי דרך בתוכנית העבודה הפיסית שהוא מייצר. כמובן שהוא יבצע קיצורי דרך אך ורק אם יוכל להבטיח שהתוצאה שתתקבל תהיה הנכונה – במילים אחרות, תוצאה זהה לזו שתקבל על ידי מעקב אחר השלבים של העיבוד הלוגי. לדוגמה, כדי להשתמש באינדקס, ה–optimizer עשוי להחליט להפעיל סינון מוקדם הרבה יותר מאשר כפי שמוכתב על ידי העיבוד הלוגי.

מהסיבות המוזכרות לעיל, חשוב לבצע הבחנה ברורה בין עיבוד לוגי לעיבוד פיסי של שאילתה.

ללא עיכובים נוספים, הבה נצלול לשלבי העיבוד הלוגי של שאילתה.

שלבי עיבוד לוגי של שאילתה

סעיף זה מציג את השלבים המעורבים בעיבוד הלוגי של שאילתה. ראשית אתאר בקצרה כל שלב. לאחר מכן, בסעיפים הבאים, אתאר את השלבים בפירוט רב הרבה יותר ואיישם אותם בשאילתה לדוגמה. תוכל להשתמש בחלק זה כמקור מידע בכל פעם שתצטרך להיזכר בסדר ובמשמעות של השלבים השונים.

קטע–קוד 1–1 מכיל צורה כללית של שאילתה, כאשר לכל שלב מצורף מספר בהתאם לסדר בו כל פסוקית מעובדת לוגית.

קטע–קוד 1–1: מספור שלבי עיבוד לוגי של שאילתה

```
(8)  SELECT (9) DISTINCT (11) <TOP_specification> <select_list>
(1)  FROM <left_table>
(3)    <join_type> JOIN <right_table>
(2)      ON <join_condition>
(4)  WHERE <where_condition>
(5)  GROUP BY <group_by_list>
(6)  WITH {CUBE | ROLLUP}
(7)  HAVING <having_condition>
(10) ORDER BY <order_by_list>
```

הפן הבולט הראשון של SQL השונה משפות תכנות אחרות הוא הסדר בו מעובד הקוד. במרבית שפות התכנות, הקוד מעובד בסדר שבו הוא נכתב. ב–SQL, הפסוקית הראשונה

שעוברת עיבוד היא הפסוקית FROM, בעוד שהפסוקית SELECT, המופיעה ראשונה, מעובדת כמעט אחרונה.

כל שלב מייצר טבלה וירטואלית המשמשת כקלט לשלב הבא. טבלאות וירטואליות אלו אינן זמינות ללקוח (יישום לקוח או שאילתה חיצונית). אך ורק הטבלה המיוצרת על ידי השלב הסופי מוחזרת ללקוח. אם פסוקית מסוימת אינה נכללת בשאילתה, העיבוד מדלג על השלב הקשור בפסוקית. כעת אתאר בקצרה את השלבים הלוגיים השונים המיושמים הן ב– SQL Server 2000 והן ב– SQL Server 2005. בהמשך הפרק אדון בנפרד בשלבים שנוספו ב– SQL Server 2005.

תיאור קצר של שלבי העיבוד הלוגי של שאילתה

אל תדאג אם תיאור השלבים לא ייראה כל–כך הגיוני כעת. תיאור זה נכתב כמקור מידע. הסעיפים שיבואו לאחר השאילתה לדוגמה יכסו את השלבים הללו בצורה מעמיקה הרבה יותר.

1. **FROM:** מכפלה קרטזית (cross join) מבוצעת בין שתי הטבלאות הראשונות בפסוקית FROM, וכתוצאה מכך נוצרת טבלה וירטואלית VT1.
2. **ON:** מסנן ON מיושם על VT1. רק שורות אשר עבורן תנאי join (<join_condition>) הוא TRUE נכנסות לטבלה VT2.
3. **OUTER (join):** אם בשאילתה נכלל OUTER JOIN (בניגוד ל– CROSS JOIN או ל– INNER JOIN), שורות מהטבלה השמורה (או מהטבלאות השמורות) אשר עבורן לא נמצאה התאמה מתווספות לטבלה VT2 כשורות חיצוניות. טבלה VT3 נוצרת. אם בפסוקית FROM מופיעות יותר משתי טבלאות, שלבים 1 עד 3 מיושמים שוב ושוב בין התוצאה של ה–join האחרון לבין הטבלה הבאה בפסוקית FROM, עד שכל הטבלאות מעובדות.
4. **WHERE:** מסנן WHERE מיושם על VT3. רק שורות אשר עבורן תנאי where (<where_condition>) הוא TRUE נכנסות לטבלה VT4.
5. **GROUP BY:** השורות מטבלה VT4 מסודרות בקבוצות בהתאם לרשימת הטורים המצוינת בפסוקית GROUP BY. טבלה VT5 נוצרת.
6. **CUBE | ROLLUP:** קבוצות–על (קבוצות של קבוצות) מתווספות לשורות מטבלה VT5. טבלה VT6 נוצרת.
7. **HAVING:** מסנן HAVING מיושם על VT6. רק קבוצות אשר עבורן תנאי having (<having_condition>) הוא TRUE נכנסות לטבלה VT7.
8. **SELECT:** רשימת SELECT מעובדת וטבלה VT8 נוצרת.
9. **DISTINCT:** שורות כפולות מוסרות מטבלה VT8. טבלה VT9 נוצרת.

10. **ORDER BY:** השורות מטבלה VT9 ממוינות לפי רשימת הטורים שמוגדרת בפסוקית ORDER BY. נוצר סמן (VC10).

11. **TOP:** מספר או אחוז השורות שהוגדר נבחר מתחילת הסמן VC10. טבלה VT11 נוצרת ומוחזרת ללקוח.

שאילתה לדוגמה מבוססת על לקוחות/הזמנות

כדי לתאר את שלבי העיבוד הלוגי בפירוט, אעקוב יחד אתך אחר מהלך של שאילתה לדוגמה. ראשית הרץ את הקוד המוצג בקטע–קוד 2–1 כדי ליצור את הטבלאות Customers ו–Orders ולהכניס בהן נתונים לדוגמה. טבלאות 1–1 ו–2–1 מציגות את התוכן של Customers ושל Orders.

קטע–קוד 2–1: שפה להגדרת נתונים (DDL) ונתונים לדוגמה עבור Customers ו–Orders

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
GO
IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
GO
CREATE TABLE dbo.Customers
(
  customerid  CHAR(5)     NOT NULL PRIMARY KEY,
  city        VARCHAR(10) NOT NULL
);

INSERT INTO dbo.Customers(customerid, city) VALUES('FISSA', 'Madrid');
INSERT INTO dbo.Customers(customerid, city) VALUES('FRNDO', 'Madrid');
INSERT INTO dbo.Customers(customerid, city) VALUES('KRLOS', 'Madrid');
INSERT INTO dbo.Customers(customerid, city) VALUES('MRPHS', 'Zion');

CREATE TABLE dbo.Orders
(
  orderid    INT         NOT NULL PRIMARY KEY,
  customerid CHAR(5)     NULL     REFERENCES Customers(customerid)
);
```

```
INSERT INTO dbo.Orders(orderid, customerid) VALUES(1, 'FRNDO');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(2, 'FRNDO');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(3, 'KRLOS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(4, 'KRLOS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(5, 'KRLOS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(6, 'MRPHS');
INSERT INTO dbo.Orders(orderid, customerid) VALUES(7, NULL);
```

טבלה 1–1: תוכן של טבלת Customers

customerid	*city*
FISSA	Madrid
FRNDO	Madrid
KRLOS	Madrid
MRPHS	Zion

טבלה 1–2: תוכן של טבלת Orders

orderid	*customerid*
1	FRNDO
2	FRNDO
3	KRLOS
4	KRLOS
5	KRLOS
6	MRPHS
7	NULL

לצורך הדגמה אשתמש בשאילתה המוצגת בקטע–קוד 1–3. השאילתה מחזירה לקוחות מהעיר מדריד אשר ביצעו פחות משלוש הזמנות (כולל אפס הזמנות), ואת מספר ההזמנות שביצעו. התוצאה ממוינת לפי מספר הזמנות, מהנמוך לגבוה. הפלט של שאילתה זו מוצג בטבלה 1–3.

קטע–קוד 3–1: שאילתה – לקוחות מדריד שלהם פחות משלוש הזמנות

```
SELECT C.customerid, COUNT(O.orderid) AS numorders
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.customerid = O.customerid
WHERE C.city = 'Madrid'
GROUP BY C.customerid
HAVING COUNT(O.orderid) < 3
ORDER BY numorders;
```

טבלה 3–1: פלט – לקוחות מדריד שלהם פחות משלוש הזמנות

customerid	*numorders*
FISSA	0
FRNDO	2

הן FISSA והן FRNDO הם לקוחות מהעיר מדריד אשר ביצעו פחות משלוש הזמנות. בחן את השאילתה ונסה לקרוא אותה תוך כדי מעקב אחר השלבים המתוארים בקטע–קוד 1–1 והסעיף "תיאור קצר של שלבי העיבוד הלוגי של שאילתה". אם זו הפעם הראשונה שהנך חושב על שאילתה במונחים אלו, ייתכן שהדבר יהיה מבלבל. הסעיף הבא יסייע לך להיכנס לעובי הקורה.

פירוט שלבי העיבוד הלוגי של שאילתה

סעיף זה מתאר את שלבי העיבוד הלוגי של שאילתה בפירוט על ידי יישומם בשאילתה לדוגמה.

שלב 1: ביצוע מכפלה קרטזית (Cross Join)

מכפלה קרטזית (cross join או join בלתי מוגבל) מבוצעת בין שתי הטבלאות הראשונות המופיעות בפסוקית FROM, וכתוצאה מכך נוצרת טבלה VT1. VT1 מכילה שורה לכל שילוב אפשרי של שורה מהטבלה השמאלית ושורה מהטבלה הימנית. אם הטבלה השמאלית מכילה n שורות והטבלה הימנית מכילה m שורות, טבלה VT1 תכיל n x m שורות. הטורים בטבלה VT1 מקבלים תחילית של שמות טבלאות המקור שלהם (או כינויי טבלה, אם הגדרת כאלה בשאילתה). בשלבים הבאים (שלב 2 ואילך), התייחסות לשם טור רב–משמעי (מופיע ביותר מטבלת קלט אחת) חייב לכלול תחילית של שם טבלה (לדוגמה,

C.customerid). אין הכרח להכליל תחילית של שם טבלה עבור טורים המופיעים בקלט אחד בלבד (למשל, O.orderid או פשוט orderid).

יישם את שלב 1 על השאילתה לדוגמה (מוצג בקטע–קוד 3–1):

```
FROM Customers AS C ... JOIN Orders AS O
```

כתוצאה תקבל את הטבלה הווירטואלית VT1 המוצגת בטבלה 4–1 עם 28 שורות (7x4).

טבלה 4–1: טבלה וירטואלית VT1 המוחזרת משלב 1

C.customerid	*C.city*	*O.orderid*	*O.customerid*
FISSA	Madrid	1	FRNDO
FISSA	Madrid	2	FRNDO
FISSA	Madrid	3	KRLOS
FISSA	Madrid	4	KRLOS
FISSA	Madrid	5	KRLOS
FISSA	Madrid	6	MRPHS
FISSA	Madrid	7	NULL
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
FRNDO	Madrid	3	KRLOS
FRNDO	Madrid	4	KRLOS
FRNDO	Madrid	5	KRLOS
FRNDO	Madrid	6	MRPHS
FRNDO	Madrid	7	NULL
KRLOS	Madrid	1	FRNDO
KRLOS	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
KRLOS	Madrid	6	MRPHS
KRLOS	Madrid	7	NULL
MRPHS	Zion	1	FRNDO

C.customerid	C.city	O.orderid	O.customerid
MRPHS	Zion	2	FRNDO
MRPHS	Zion	3	KRLOS
MRPHS	Zion	4	KRLOS
MRPHS	Zion	5	KRLOS
MRPHS	Zion	6	MRPHS
MRPHS	Zion	7	NULL

שלב 2: יישום המסנן ON (Join Condition)

מסנן ON הוא הראשון משלושה מסננים אפשריים (ON, WHERE ו–HAVING) אשר ניתן להגדיר בשאילתה. הביטוי הלוגי במסנן ON מיושם על כל השורות בטבלה הווירטואלית המוחזרת על ידי השלב הקודם (VT1). רק שורות שעבורן תנאי join הוא TRUE הופכות לחלק מהטבלה הווירטואלית המוחזרת על ידי השלב הנוכחי (VT2).

לוגיקת שלושת הערכים

אבקש לסטות מעט מהנושא כדי לכסות היבטים חשובים של SQL הקשורים לשלב זה. הערכים האפשריים של ביטוי לוגי ב–SQL הם TRUE, FALSE ו–UNKNOWN ומכאן השם לוגיקת שלושת–הערכים. לוגיקת שלושת–הערכים היא ייחודית ל–SQL. ביטויים לוגיים במרבית שפות התכנות יכולים להיות רק TRUE או FALSE. הערך הלוגי UNKNOWN ב–SQL מופיע בדרך–כלל בביטוי לוגי שבו מעורב NULL (למשל, הערך הלוגי של כל אחד משלושת הביטויים הבאים הוא UNKNOWN: X + NULL > Y ;NULL = NULL ;NULL > 42). הערך הייחודי NULL מייצג בדרך–כלל ערך חסר או לא רלוונטי. כאשר משווים ערך חסר לערך אחר (אפילו ל–NULL אחר), התוצאה הלוגית היא UNKNOWN.

הטיפול בתוצאות לוגיות שהן UNKNOWN יכול להיות מאוד מבלבל. בעוד ש– NOT TRUE הוא FALSE, ו– NOT FALSE הוא TRUE, ההופכי של UNKNOWN (NOT UNKNOWN) הוא עדיין UNKNOWN.

תוצאות לוגיות שהן UNKNOWN וכן NULLs מטופלים בצורה לא עקבית באלמנטים שונים של השפה. למשל, כל מסנני השאילתה (ON, WHERE ו–HAVING) מתייחסים ל–UNKNOWN בצורה דומה ל–FALSE. שורה אשר עבורה מסנן הוא UNKNOWN נמחקת מהתוצאה. מצד שני, ערך UNKNOWN באילוץ CHECK מטופל למעשה

כ–TRUE. נניח שיש לך בטבלה אילוץ CHECK הדורש שטור השכר יהיה גדול מאפס. שורה אשר מוכנסת לטבלה עם שכר NULL מתקבלת, מכיוון ש– (NULL > 0) הוא UNKNOWN, והוא מטופל כ–TRUE באילוץ CHECK.

השוואה בין שני NULL במסננים מניבה UNKNOWN, אשר כפי שהזכרתי קודם מטופל כ–FALSE – כאילו NULL אחד שונה מהשני.

מצד שני, אילוץ UNIQUE, מיון וקיבוץ מתייחסים ל–NULLs כשווים:

- אינך יכול להכניס לטבלה שתי שורות עם NULL בטור שמוגדר לו אילוץ UNIQUE.
- פסוקית GROUP BY מקבצת את כל ה–NULLs לקבוצה אחת.
- פסוקית ORDER BY ממיינת את כל ה–NULLs יחד.

בקיצור, כדי לחסוך לעצמך עוגמת נפש, כדאי שתהיה מודע לדרך בה אלמנטים שונים של השפה מטפלים בתוצאות לוגיות של UNKNOWN וב–NULLs.

יישם את שלב 2 על השאילתה לדוגמה:

```
ON C.customerid = O.customerid
```

טבלה 5–1 מציגה את הערך של הביטוי הלוגי במסנן ON עבור השורות מטבלה VT1.

טבלה 5–1: תוצאות לוגיות של מסנן ON מיושמות על שורות מטבלה VT1

Match?	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
FALSE	FISSA	Madrid	1	FRNDO
FALSE	FISSA	Madrid	2	FRNDO
FALSE	FISSA	Madrid	3	KRLOS
FALSE	FISSA	Madrid	4	KRLOS
FALSE	FISSA	Madrid	5	KRLOS
FALSE	FISSA	Madrid	6	MRPHS
UNKNOWN	FISSA	Madrid	7	NULL
TRUE	FRNDO	Madrid	1	FRNDO
TRUE	FRNDO	Madrid	2	FRNDO
FALSE	FRNDO	Madrid	3	KRLOS

Match?	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
FALSE	FRNDO	Madrid	4	KRLOS
FALSE	FRNDO	Madrid	5	KRLOS
FALSE	FRNDO	Madrid	6	MRPHS
UNKNOWN	FRNDO	Madrid	7	NULL
FALSE	KRLOS	Madrid	1	FRNDO
FALSE	KRLOS	Madrid	2	FRNDO
TRUE	KRLOS	Madrid	3	KRLOS
TRUE	KRLOS	Madrid	4	KRLOS
TRUE	KRLOS	Madrid	5	KRLOS
FALSE	KRLOS	Madrid	6	MRPHS
UNKNOWN	KRLOS	Madrid	7	NULL
FALSE	MRPHS	Zion	1	FRNDO
FALSE	MRPHS	Zion	2	FRNDO
FALSE	MRPHS	Zion	3	KRLOS
FALSE	MRPHS	Zion	4	KRLOS
FALSE	MRPHS	Zion	5	KRLOS
TRUE	MRPHS	Zion	6	MRPHS
UNKNOWN	MRPHS	Zion	7	NULL

רק שורות אשר עבורן תנאי join הוא TRUE נכנסות ל–VT2 – טבלת הקלט הווירטואלית של השלב הבא, המוצגת בטבלה 6–1.

טבלה 6–1: טבלה וירטואלית VT2 המוחזרת משלב 2

Match?	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
TRUE	FRNDO	Madrid	1	FRNDO
TRUE	FRNDO	Madrid	2	FRNDO
TRUE	KRLOS	Madrid	3	KRLOS
TRUE	KRLOS	Madrid	4	KRLOS
TRUE	KRLOS	Madrid	5	KRLOS
TRUE	MRPHS	Zion	6	MRPHS

שלב 3: הוספת שורות חיצוניות

שלב זה רלוונטי רק עבור outer join. עבור outer join, אתה מסמן את אחת מטבלאות הקלט או את שתיהן כטבלה שמורה על ידי ציון סוג ה– outer join (LEFT, RIGHT, או FULL). המשמעות של סימון טבלה כשמורה היא שאתה מעוניין שכל שורותיה יוחזרו, אפילו אם הן סוננו החוצה על ידי תנאי ה–join. Left outer join מסמן את הטבלה השמאלית כשמורה, right outer join מסמן את הימנית, ו– full outer join מסמן את שתיהן. שלב 3 מחזיר את השורות מטבלה VT2 וכן שורות מהטבלה השמורה אשר עבורן לא נמצאה התאמה בשלב 2. שורות נוספות אלו נקראות שורות חיצוניות. במאפיינים (ערכי טור) של השורות החיצוניות השייכים לטבלאות הלא שמורות יופיעו ערכי NULL. כתוצאה נוצרת טבלה וירטואלית VT3.

בדוגמה שלנו, הטבלה השמורה היא Customers:

```
Customers AS C LEFT OUTER JOIN Orders AS O
```

רק הלקוח FISSA לא מצא הזמנה תואמת (לא היה חלק מטבלה VT2). לפיכך, FISSA מתווסף לשורות מהשלב הקודם עם NULLs במאפייני Orders וכתוצאה מכך נוצרת טבלה וירטואלית VT3 (מוצגת בטבלה 7–1).

טבלה 7–1: טבלה וירטואלית VT3 מוחזרת משלב 3

C.customerid	*C.city*	*O.orderid*	*O.customerid*
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
MRPHS	Zion	6	MRPHS
FISSA	Madrid	NULL	NULL

שים לב: אם יותר משתי טבלאות מאוחדות, שלבים 1 עד 3 ייושמו בין VT3 לבין הטבלה השלישית מהפסוקית FROM. תהליך זה יחזור על עצמו אם טבלאות נוספות מופיעות בפסוקית FROM, והטבלה הווירטואלית הסופית תשמש כקלט לשלב הבא.

שלב 4: יישום המסנן WHERE

מסנן WHERE מיושם על כל השורות בטבלה הווירטואלית שהוחזרה מהשלב הקודם. רק שורות שעבורן תנאי where (<where condition>) הוא TRUE הופכות לחלק מהטבלה הווירטואלית המוחזרת משלב זה (VT4).

אזהרה: מכיוון שהנתונים אינם מקובצים עדיין, אינך יכול להשתמש עדיין במסנני צבירה (אגרגציה) – למשל, אינך יכול לכתוב: WHERE orderdate = MAX(orderdate). כמו כן, אינך יכול להתייחס לכינויי טור הנוצרים על ידי רשימת ה–SELECT שכן רשימת ה–SELECT עדיין לא עברה עיבוד – למשל, אינך יכול לכתוב: SELECT YEAR (orderdate) AS orderyear … WHERE orderyear > 2000.

היבט מבלבל של שאילתות המכילות פסוקית OUTER JOIN הוא האם לציין ביטוי לוגי במסנן ON או במסנן WHERE. ההבדל העיקרי בין השניים הוא ש–ON מיושם לפני הוספת שורות חיצוניות (שלב 3), בעוד ש–WHERE מיושם אחרי שלב 3. הסרת שורה מטבלה שמורה על ידי מסנן ON אינה סופית מכיוון ששלב 3 יוסיף אותה בחזרה, בעוד שהסרת שורה על ידי מסנן WHERE היא סופית. זכור עובדה זו, שתסייע לך לעשות את ההחלטה הנכונה.

למשל, נניח שאתה מעוניין להחזיר לקוחות מסוימים ואת ההזמנות שלהם מטבלאות Customers ו–Orders. הלקוחות שאתה מעוניין להחזיר הם רק לקוחות מהעיר מדריד, הן אלה שביצעו הזמנות והן אלה שלא. Outer join מיועד בדיוק לבקשות מסוג זה. אתה מבצע left outer join בין Customers ו–Orders ועל ידי כך מסמן את Customers כטבלה שמורה. כדי שתוכל להחזיר לקוחות שלא ביצעו הזמנות, עליך לציין את המתאם (קורלציה) בין לקוחות להזמנות בפסוקית ON (ON C.customerid = O.customerid). לקוחות ללא הזמנות מוסרים בשלב 2 אך מתווספים חזרה בשלב 3 כשורות חיצוניות. לעומת זאת, מכיוון שהנך מעוניין להשאיר רק שורות עבור לקוחות מהעיר מדריד, בין אם ביצעו הזמנות ובין אם לאו, עליך לציין את מסנן העיר בפסוקית WHERE (WHERE C.city='Madrid'). ציון מסנן העיר בפסוקית ON יגרום לכך שלקוחות שאינם ממדריד יתווספו לתוצאה על ידי שלב 3.

טיפ: יש הבדל לוגי בין הפסוקיות ON ו–WHERE רק בעת השימוש ב– outer join. כאשר משתמשים ב– inner join, אין זה משנה היכן אתה מציין את הביטויים הלוגיים שלך מכיוון שהעיבוד מדלג על שלב 3. המסננים מיושמים אחד אחר השני ללא שלב ביניים ביניהם. ישנו יוצא דופן אחד הרלוונטי רק כאשר משתמשים באפשרות של GROUP BY ALL. אדון באפשרות זו בסעיף הבא, המכסה את שלב GROUP BY.

יישם את המסנן על השאילתה לדוגמה:

```
WHERE C.city = 'Madrid'
```

השורה עבור הלקוח MRPHS מטבלה VT3 מוסרת מכיוון שאינו ממדריד, ונוצרת הטבלה הווירטואלית VT4, המוצגת בטבלה 8–1.

טבלה 8–1: טבלה וירטואלית VT4 המוחזרת משלב 4

C.customerid	*C.city*	*O.orderid*	*O.customerid*
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
FISSA	Madrid	NULL	NULL

שלב 5: קיבוץ

השורות מהטבלה המוחזרת על ידי השלב הקודם מסודרות בקבוצות. כל צירוף ייחודי של ערכים ברשימת הטורים המופיעה בפסוקית GROUP BY יוצר קבוצה. כל שורת בסיס מהשלב הקודם משובצת לקבוצה אחת ויחידה. טבלה וירטואלית VT5 נוצרת. VT5 מורכבת משתי יחידות: יחידת הקבוצות המורכבת מהקבוצות עצמן, ויחידת המידע הגולמי המורכבת משורות הבסיס המצורפות מהשלב הקודם.

יישם את שלב 5 על השאילתה לדוגמה:

```
GROUP BY C.customerid
```

מתקבלת הטבלה הווירטואלית VT5 המוצגת בטבלה 9–1.

טבלה 9–1: טבלה וירטואלית VT5 המוחזרת משלב 5

Groups	*Raw*			
C.customerid	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
FRNDO	FRNDO	Madrid	1	FRNDO
	FRNDO	Madrid	2	FRNDO

Groups	*Raw*			
C.customerid	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
KRLOS	KRLOS	Madrid	3	KRLOS
	KRLOS	Madrid	4	KRLOS
	KRLOS	Madrid	5	KRLOS
FISSA	FISSA	Madrid	NULL	NULL

אם פסוקית GROUP BY מצוינת בשאילתה, כל השלבים הבאים (HAVING, SELECT וכו') יכולים לציין רק ביטויים שתוצאתם היא ערך סקלרי (יחידני) עבור קבוצה. במילים אחרות, התוצאות יכולות להיות או טור/ביטוי המשתתף ברשימת ה– GROUP BY, למשל, C.customerid, או פונקציית צבירה (אגרגציה) כגון COUNT(O.orderid). ההיגיון מאחורי מגבלה זו הוא שבתוצאה הסופית תיווצר שורה יחידה לכל קבוצה (אלא אם כן היא סוננה החוצה). בחן את VT5 בטבלה 9–1, ונסה לחשוב מה אמורה השאילתה להחזיר עבור הלקוח FRNDO אם רשימת ה–SELECT שציינת הייתה SELECT C.customerid, O.orderid. ישנם שני ערכי orderid שונים בקבוצה; לפיכך, התשובה איננה דטרמיניסטית. SQL אינו מאפשר בקשה כזו. מצד שני, אם תציין: SELECT C.customerid, COUNT(O.orderid) AS numorders, התשובה עבור FRNDO תהיה דטרמיניסטית: 2.

שים לב: מותר לך גם לקבץ לפי התוצאה של ביטוי – למשל, GROUP BY YEAR(orderdate). אם תעשה כך, כשאתה עובד ב– SQL Server 2000, כל השלבים הבאים לא יוכלו לבצע כל מניפולציה נוספת לביטוי ה– GROUP BY, אלא אם כן זהו טור בסיס. למשל, הבקשה הבאה אינה מותרת ב– SQL Server 2000:

```
SELECT YEAR(orderdate) + 1 AS nextyear … GROUP BY YEAR(orderdate)
```

ב– SQL Server 2005 מגבלה זו הוסרה.

שלב זה מתייחס ל–NULLs כשווים. כלומר, כל ה–NULLs מקובצים לקבוצה אחת ממש כמו ערך ידוע.

כפי שציינתי קודם, הקלט לשלב GROUP BY הוא הטבלה הווירטואלית המוחזרת על ידי השלב הקודם (VT4). אם ציינת GROUP BY ALL, קבוצות אשר הוסרו על ידי השלב הרביעי (מסנן WHERE) מתווספות לטבלת התוצאה הווירטואלית של שלב זה (VT5) עם סט ריק ביחידת המידע הגולמי. זהו המקרה היחיד בו יש הבדל בין ציון ביטוי לוגי בפסוקית ON או בפסוקית WHERE כאשר משתמשים ב– inner join. אם תשנה את הדוגמה שלנו כך שתשתמש ב– GROUP BY ALL C.customerid במקום ב– GROUP BY C.customerid, תמצא שהלקוח MRPHS, שהוסר על ידי מסנן WHERE,

יתווסף ליחידת הקבוצות של VT5, בצירוף סט ריק ביחידת המידע הגולמי. פונקציית הצבירה COUNT באחד מהשלבים הבאים תהיה אפס עבור קבוצה כזו, בעוד שכל פונקציות הצבירה האחרות (SUM, AVG, MIN, MAX) יהיו NULL.

שים לב: האפשרות GROUP BY ALL היא מאפיין לא סטנדרטי, שנשאר במוצר מסיבות היסטוריות. הוא מציג סוגיות סמנטיות רבות כאשר מיקרוסופט מוסיפה מאפייני T-SQL חדשים. על אף שמאפיין זה נתמך לחלוטין ב– SQL Server 2005, ייתכן שתעדיף להימנע משימוש בו מכיוון שמיקרוסופט עשויה להפסיק לתמוך בפסוקית זו בעתיד.

שלב 6: יישום האפשרות CUBE או ROLLUP

אם מצוין CUBE או ROLLUP, נוצרות קבוצות–על ומתווספות לקבוצות בטבלה הווירטואלית המוחזרת מהשלב הקודם. טבלה וירטואלית VT6 נוצרת.

בדוגמה שלנו נדלג על שלב 6 מכיוון ש–CUBE ו–ROLLUP אינם מופיעים בשאילתה לדוגמה. CUBE ו–ROLLUP יכוסו בשלב מאוחר יותר בספר בפרק 6.

שלב 7: יישום המסנן HAVING

מסנן HAVING מיושם על הקבוצות בטבלה המוחזרת מהשלב הקודם. רק קבוצות שעבורן תנאי having (<having_condition>) הוא TRUE הופכות לחלק מהטבלה הווירטואלית המוחזרת על ידי שלב זה (VT7). מסנן HAVING הוא המסנן הראשון והיחיד המתייחס לנתונים המקובצים.

יישם שלב זה על השאילתה לדוגמה:

```
HAVING COUNT(O.orderid) < 3
```

הקבוצה של לקוח KRLOS מוסרת מכיוון שהיא מכילה שלוש הזמנות. נוצרת טבלה וירטואלית VT7, המוצגת בטבלה 10–1.

טבלה 10–1: טבלה וירטואלית VT7 המוחזרת על ידי שלב 7

C.customerid	*C.customerid*	*C.city*	*O.orderid*	*O.customerid*
FRNDO	FRNDO	Madrid	1	FRNDO
	FRNDO	Madrid	2	FRNDO
FISSA	FISSA	Madrid	NULL	NULL

שים לב: חשוב לציין כאן COUNT(O.orderid) ולא COUNT(*). מכיוון שה–join הוא outer join, שורות חיצוניות התווספו עבור לקוחות ללא הזמנות. COUNT(*) היה מצרף שורות חיצוניות לספירה תוך כדי הוספה שגויה של הזמנה אחת ללקוח FISSA, לו אין הזמנות. COUNT(O.orderid) סופר בצורה נכונה את מספר ההזמנות לכל לקוח, ומחשב 0 הזמנות ללקוח FISSA. זכור ש– COUNT(<expression>) מתעלם מ–NULLs כמו כל פונקציית צבירה אחרת.

פונקציית צבירה אינה מקבלת תת–שאילתה כקלט – למשל,

HAVING SUM((SELECT …)) > 10

שלב 8: עיבוד רשימת ה–SELECT

על אף שרשימת ה–SELECT מצוינת ראשונה בשאילתה, היא מעובדת רק בשלב השמיני. שלב ה–SELECT בונה את הטבלה אשר בסופו של דבר תוחזר ללקוח. הביטויים ברשימת ה–SELECT יכולים להחזיר טורי בסיס ומניפולציות של טורי בסיס מהטבלה הווירטואלית המוחזרת מהשלב הקודם. זכור שאם השאילתה היא שאילתת צבירה, לאחר שלב 5 תוכל להתייחס לטורי בסיס מהשלב הקודם אך ורק אם הם חלק מיחידת הקבוצות (רשימת GROUP BY). אם תרצה להתייחס לטורים מיחידת המידע הגולמי, אלו חייבים להופיע בתוך פונקציית הצבירה. טורי בסיס הנבחרים מהשלב הקודם שומרים על שמות הטורים שלהם אלא אם כן תיתן להם כינויים (למשל, col1 AS c1). ביטויים שאינם טורי בסיס חייבים לקבל כינוי כדי שיהיה להם שם טור בטבלת התוצאה – למשל, YEAR(orderdate) AS orderyear.

חשוב: לא ניתן להשתמש בכינויים הנוצרים על ידי רשימת ה–SELECT בשלבים קודמים. למעשה, בכינויי ביטויים לא ניתן אפילו להשתמש על ידי ביטויים אחרים בתוך אותה רשימת SELECT. ההיגיון מאחורי מגבלה זו הוא עוד פן ייחודי של SQL, היותו פעולה בו–זמנית. למשל, ברשימת ה–SELECT הבאה, הסדר הלוגי לפיו הביטויים מוערכים אינו חשוב ואינו מובטח: SELECT c1 + 1 AS e1, c2 + 1 AS e2. לפיכך, רשימת ה–SELECT הבאה אינה נתמכת: SELECT c1 + 1 AS e1, e1 + 1 AS e2. מותר לך להשתמש שוב בכינויי טורים רק בשלבים הבאים אחרי רשימת ה–SELECT, כמו שלב ה– ORDER BY – למשל, SELECT YEAR(orderdate) AS orderyear … ORDER BY orderyear.

יישם שלב זה על השאילתה לדוגמה:

```
SELECT C.customerid, COUNT(O.orderid) AS numorders
```

מתקבלת טבלה וירטואלית VT8, המוצגת בטבלה 11–1.

טבלה 11–1: טבלה וירטואלית VT8 המוחזרת משלב 8

C.customerid	*numorders*
FRNDO	2
FISSA	0

המושג פעולה בו–זמנית עשוי להיות קשה לתפיסה. למשל, במרבית סביבות הפיתוח, לצורך החלפת ערכים בין משתנים תשתמש במשתנה זמני. לעומת זאת, ב–SQL כדי להחליף ערכים בין טורים בטבלה תוכל להשתמש ב:

```
UPDATE dbo.T1 SET c1 = c2, c2 = c1;
```

לוגית, עליך להניח שכל הפעולה מבוצעת בו–זמנית. כאילו הטבלה אינה מעודכנת עד שכל הפעולה מסתיימת ואז התוצאה מחליפה את המקור. מסיבות דומות, UPDATE זה:

```
UPDATE dbo.T1 SET c1 = c1 + (SELECT MAX(c1) FROM dbo.T1);
```

יעדכן את כל השורות של T1 ויוסיף ל–c1 את הערך המקסימלי מ–T1 כאשר העדכון מתחיל. אינך צריך לדאוג שמא הערך המקסימלי של c1 ימשיך להשתנות כאשר הפעולה תמשיך, מכיוון שהפעולה כאילו מתרחשת בו–זמנית.

שלב 9: יישום הפסוקית DISTINCT

אם בשאילתה מצוינת פסוקית DISTINCT, שורות כפולות מוסרות מהטבלה הווירטואלית המוחזרת על ידי השלב הקודם, וטבלה וירטואלית VT9 נוצרת.

בדוגמה שלנו נדלג על שלב 9 מכיוון שבשאילתה לדוגמה לא מופיע DISTINCT. למעשה, DISTINCT היא פסוקית מיותרת כאשר משתמשים ב– GROUP BY והיא לא תסיר אף שורה.

שלב 10: יישום הפסוקית ORDER BY

השורות מהשלב הקודם ממוינות לפי רשימת הטורים המוגדרת בפסוקית ORDER BY ומוחזרות בסמן (cursor) VC10. שלב זה הוא הראשון והיחיד בו ניתן לשוב ולהשתמש בכינויי טורים שנוצרו ברשימת ה–SELECT.

הן לפי ANSI SQL:1992 והן לפי ANSI SQL:1999, אם מצוין DISTINCT, לביטויים בפסוקית ORDER BY יש גישה רק לטבלה הווירטואלית המוחזרת על ידי השלב הקודם

(VT9). כלומר, ניתן למיין רק לפי מה שבחרת. ANSI SQL:1992 מציב את אותה מגבלה אפילו כאשר DISTINCT אינו מצוין. לעומת זאת, ANSI SQL:1999 מרחיב את התמיכה ב– ORDER BY על ידי כך שהוא מאפשר גישה הן לטבלת הקלט הווירטואלית של שלב ה–SELECT והן לטבלת הפלט הווירטואלית של השלב הנוכחי. כלומר, אם DISTINCT אינו מופיע, ניתן לציין בפסוקית ORDER BY כל ביטוי שהיה מותר בפסוקית SELECT. בפרט, ניתן למיין לפי ביטויים שאינם מוחזרים בתוצאה הסופית.

ישנו היגיון באיסור גישה לביטויים שאינך מחזיר כאשר DISTINCT מצוין. כאשר מוסיפים ביטויים לרשימת ה–SELECT, קיימת אפשרות ש–DISTINCT ישנה את מספר השורות המוחזרות. ללא DISTINCT, כמובן ששינויים ברשימת ה–SELECT לא משפיעים על מספר השורות המוחזרות. T-SQL תמיד יישם את הגישה של ANSI SQL:1999.

בדוגמה שלנו, מכיוון ש–DISTINCT אינו מצוין, לפסוקית ORDER BY יש גישה הן לטבלה VT7, המוצגת בטבלה 1–10, והן לטבלה VT8, המוצגת בטבלה 1–11.

בפסוקית ORDER BY ניתן לציין גם מיקום סידורי של טורי תוצאה מרשימת ה–SELECT. לדוגמה, השאילתה הבאה ממיינת את ההזמנות ראשית לפי customerid ואז לפי orderid:

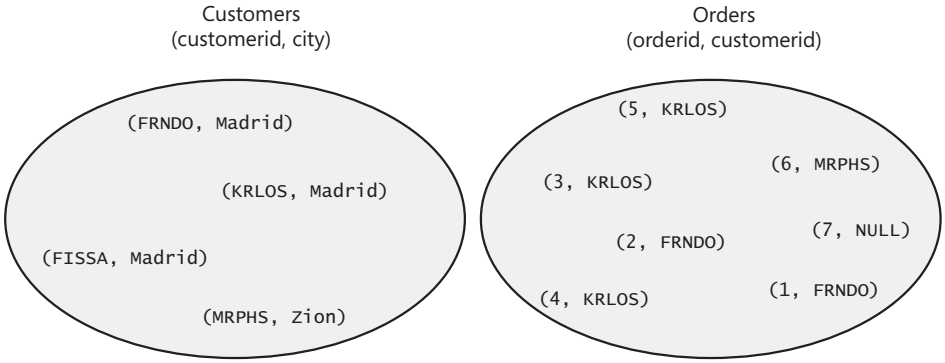
```
SELECT orderid, customerid FROM dbo.Orders ORDER BY 2, 1;
```

אף על פי כן, נוהג כזה אינו מומלץ מכיוון שאתה עלול לבצע שינויים ברשימת ה–SELECT ולשכוח לעדכן את רשימת ה– ORDER BY בהתאם. כמו כן, כאשר מחרוזת השאילתה ארוכה, קשה להבחין איזה פריט מרשימת ה– ORDER BY מתייחס לאיזה פריט מרשימת ה–SELECT.

חשוב: שלב זה שונה מכל השלבים האחרים במובן זה שאינו מחזיר טבלה; במקום זאת, הוא מחזיר סמן. זכור ש–SQL מבוסס על תורת הקבוצות (set theory). לשורות בסט אין סדר קבוע מראש; זהו אוסף לוגי של איברים, ללא כל חשיבות לסדר ביניהם. שאילתה המיישמת מיון לשורות בטבלה מחזירה אובייקט המכיל שורות מסודרות בסדר פיסי מסוים. ANSI קוראת לאובייקט כזה **cursor (סמן)**. הבנת שלב זה הינה אחד הדברים העקרוניים ביותר בהבנה נכונה של SQL.

בדרך כלל, כאשר מתארים תוכן של טבלה, מרבית האנשים (ואני ביניהם) משרטטים מבלי משים את השורות בסדר מסוים. למשל, מוקדם יותר הצגתי את טבלאות 1–1 ו–1–2 כדי לתאר את התוכן של טבלאות Customers ו–Orders. על ידי הצגת השורות בזו אחר זו, גרמתי, מבלי להתכוון, לבלבול–מה, בכך שרמזתי על סדר מסוים. דרך נכונה יותר לשרטט את התוכן של טבלאות Customers ו–Orders תהיה זו המוצגת בתרשים 1–1.

תרשים 1–1: סטים Customers ו–Orders



שים לב: אף על פי ששפת SQL לא מניחה שום סדר נתון לשורות בטבלה, היא כן מחזיקה מיקום סידורי עבור טורים, המבוסס על סדר היצירה שלהם. ציון * SELECT (על אף שזהו נוהג שאינו מומלץ ממספר סיבות שאתאר בהמשך הספר) מבטיח שהטורים יוחזרו לפי סדר יצירתם.

מכיוון ששלב זה אינו מחזיר טבלה (הוא מחזיר סמן), שאילתה עם פסוקית ORDER BY אינה יכולה לשמש כביטוי טבלה – כלומר, view, inline table-valued function, תת–שאילתה, טבלה נגזרת, או ביטוי טבלה שגור (CTE). התוצאה צריכה להיות מוחזרת ליישום הלקוח המצפה לקבל בחזרה קבוצת רשומות פיסית. למשל, שאילתת הטבלה הנגזרת הבאה אינה חוקית ומייצרת שגיאה:

```
SELECT *
FROM (SELECT orderid, customerid
      FROM dbo.Orders
      ORDER BY orderid) AS D;
```

בדומה, ה–view הבא אינו חוקי:

```
CREATE VIEW dbo.VSortedOrders
AS

SELECT orderid, customerid
FROM dbo.Orders
ORDER BY orderid
GO
```

ב–SQL שאילתה עם פסוקית ORDER BY אינה מותרת בביטוי טבלה. ב–T-SQL יש יוצא דופן לכלל זה המתואר בשלב הבא – יישום אפשרות TOP.

אם כך זכור, אל תניח שום סדר נתון לשורות בטבלה. והפוך, אל תציין פסוקית ORDER BY אלא אם כן אתה באמת צריך את השורות ממוינות. למיון יש מחיר – על SQL Server לבצע סריקה ממוינת של אינדקס או לחליפין ליישם אופרטור של מיון.

שלב ה– ORDER BY מתייחס ל–NULLs כשווים. כלומר, NULLs ממוינים יחד. ANSI משאירה את השאלה האם NULLs ממוינים לפני או אחרי ערכים ידועים, למוצר הממ‌מש, אשר חייב להיות עקבי. T-SQL ממיין NULLs לפני ערכים ידועים.

יישם שלב זה על השאילתה לדוגמה:

```
ORDER BY numorders
```

מתקבל הסמן VC10 המוצג בטבלה 12–1.

טבלה 12–1: סמן VC10 מוחזר משלב 10

C.customerid	*numorders*
FISSA	0
FRNDO	2

שלב 11: יישום האפשרות TOP

אפשרות TOP מאפשרת לך לציין מספר או אחוז של שורות (מעוגל כלפי מעלה) שיוחזרו. ב– SQL Server 2000 הקלט ל–TOP חייב להיות קבוע, בעוד שב– SQL Server 2005 הקלט יכול להיות כל ביטוי עצמאי. מספר השורות שהוגדר נבחר מתחילת הסמן שהוחזר על ידי השלב הקודם. טבלה VT11 נוצרת ומוחזרת ללקוח.

שים לב: האפשרות TOP היא ספציפית ל–T-SQL ואינה רלציונית.

שלב זה נשען על הסדר הפיסי של השורות כדי לקבוע אילו שורות נחשבות למספר השורות ה"ראשונות" המבוקש. אם בשאילתה מצוינת פסוקית ORDER BY עם רשימת ORDER BY ייחודית, התוצאה היא דטרמיניסטית. כלומר, קיימת רק תוצאה נכונה אחת, המכילה את מספר השורות הראשונות המבוקש בהתבסס על המיון שהוגדר. בדומה, כאשר מצוינת פסוקית ORDER BY עם רשימת ORDER BY שאינה ייחודית אך עם אפשרות TOP מצוין WITH TIES, התוצאה דטרמיניסטית גם כן. SQL Server בוחן את השורה האחרונה שהוחזרה פיסית ומחזיר את כל השורות האחרות מהטבלה שלהן ערך מיון זהה.

לעומת זאת, כאשר מצוינת רשימת ORDER BY שאינה ייחודית, ללא אפשרות WITH TIES, או ש– ORDER BY אינו מצוין כלל, שאילתת TOP אינה דטרמיניסטית. כלומר, השורות המוחזרות הן אלו שה– SQL Server במקרה ניגש אליהן ראשונות, ועשויות להיות תוצאות שונות שייחשבו נכונות.

אם ברצונך להבטיח דטרמיניסטיות, שאילתת TOP חייבת לכלול או רשימת ORDER BY ייחודית, או את האפשרות WITH TIES.

כפי שתוכל לשער, שאילתות TOP מכילות לרוב פסוקית ORDER BY הקובעת אילו שורות להחזיר. SQL Server מאפשר לך לציין שאילתות TOP בביטויי טבלה. אין הרבה היגיון באפשרות לציין שאילתות TOP בביטויי טבלה ללא מתן אפשרות לציין גם פסוקית ORDER BY (ראה מגבלה בשלב 10). לפיכך, שאילתות עם פסוקית ORDER BY מותרות למעשה בביטויי טבלה אך ורק אם מצוין גם TOP. במילים אחרות, שאילתה המכילה הן פסוקית TOP והן פסוקית ORDER BY מחזירה תוצאה רלציונית. האירוניה כאן היא שעל ידי שימוש באפשרות TOP שאינה תקנית ואינה רלציונית, שאילתה שאחרת הייתה מחזירה סמן, מחזירה כעת תוצאה רלציונית. תמיכה במאפיינים שאינם תקניים ואינם רלציונים (מעשיים ככל שיהיו) מאפשרת למתכנתים לנצל אותם לרעה, בדרכים אבסורדיות לעיתים, שלא היו נתמכות אחרת.

להלן דוגמה:

```
SELECT *
FROM (SELECT TOP 100 PERCENT orderid, customerid
      FROM dbo.Orders
      ORDER BY orderid) AS D;
```

או:

```
CREATE VIEW dbo.VSortedOrders
AS

SELECT TOP 100 PERCENT orderid, customerid
FROM dbo.Orders
ORDER BY orderid
GO
```

בדוגמה שלנו נדלג על שלב 11 שכן TOP אינו מופיע בשאילתה.

שלבים חדשים בעיבוד לוגי של שאילתה ב– SQL Server 2005

סעיף זה מתמקד בשלבי העיבוד המעורבים באלמנטים החדשים של שאילתות ב– SQL Server 2005. אלו כוללים אופרטורים טבלאיים (APPLY, PIVOT ו–UNPIVOT), פסוקית OVER החדשה, ופעולות סט חדשות (EXCEPT ו–INTERSECT).

שים לב: האופרטורים APPLY, PIVOT ו–UNPIVOT אינם תואמי ANSI אלא הם הרחבות ספציפיות ל–T-SQL.

אני מוצא שזה בעייתי מעט לכסות בפירוט בפרק הראשון את שלבי העיבוד הלוגי המעורבים בגרסת המוצר החדשה. אלמנטים אלו חדשים לגמרי, ויש כל–כך הרבה לומר על כל אחד מהם. במקום זאת, אביא כאן סקירה קצרה של כל אלמנט ואנהל דיונים מפורטים מאוחר יותר בספר בפרקים ממוקדים.

כפי שציינתי קודם לכן, המטרה שלי בפרק זה היא לתת לך תקציר אליו תוכל לחזור מאוחר יותר כאשר תתלבט בנוגע להיבטים הלוגיים של אלמנטים של שאילתה והדרך בה הם פועלים הדדית. עם מחשבה זו ברקע, המשמעות המלאה של שלבי העיבוד הלוגי של שאילתה, המטפלים באלמנטים החדשים עשויה להיות לא לגמרי ברורה לך כעת. אל דאגה, לאחר קריאת הפרקים הדנים בכל אלמנט בפירוט, סביר שתמצא את התקציר שאני מביא בפרק זה שימושי.

אופרטורים טבלאיים

SQL Server 2005 תומך בארבעה סוגים של אופרטורים טבלאיים בפסוקית FROM בשאילתה: JOIN, APPLY, PIVOT ו–UNPIVOT.

את שלבי העיבוד הלוגיים המעורבים ב–joins כיסיתי מוקדם יותר; כמו כן אמשיך ואדון ב–joins ביתר פירוט גם בפרק 5. כאן אתאר בקצרה את שלושת האופרטורים החדשים וכיצד הם פועלים הדדית.

אופרטורים טבלאיים מקבלים כקלטים טבלה אחת או שתיים. נקרא להם **קלט שמאל** ו**קלט ימין** בהתבסס על המיקום שלהן ביחס למילה השמורה של האופרטור הטבלאי (JOIN, APPLY, PIVOT, UNPIVOT). ממש כמו joins, כל האופרטורים הטבלאיים, מקבלים טבלה וירטואלית כקלט שמאל שלהם. האופרטור הטבלאי הראשון שמופיע בפסוקית FROM מקבל ביטוי טבלה כקלט שמאל ומחזיר טבלה וירטואלית כתוצאה. ביטוי טבלה יכול להחליף מספר דברים: טבלה אמיתית, טבלה זמנית, משתנה טבלה, טבלה נגזרת, CTE, view, או פונקציה טבלאית.

מידע נוסף: לפרטים אודות ביטויי טבלה, אנא עיין בפרק 4.

האופרטור הטבלאי השני המופיע בפסוקית FROM מקבל את הטבלה הווירטואלית המוחזרת מפעולת הטבלה הקודמת כקלט שמאל.

כל אופרטור טבלאי מאגד בתוכו סדרה שונה של צעדים. לצורך הנוחות והבהירות אוסיף למספרי הצעדים תחילית עם ראשי התיבות של האופרטור הטבלאי (J עבור JOIN, A עבור APPLY, P עבור PIVOT ו–U עבור UNPIVOT).

להלן ארבעת האופרטורים הטבלאיים בצירוף האלמנטים שלהם:

```
(J) <left_table_expression>
      <join_type> JOIN <right_table_expression>
       ON <join_condition>

(A) <left_table_expression>
      {CROSS | OUTER} APPLY <right_table_expression>

(P) <left_table_expression>
      PIVOT (<aggregate_func(<expression>)> FOR
        <source_col> IN(<target_col_list>))
        AS <result_table_alias>

(U) <left_table_expression>
      UNPIVOT (<target_values_col> FOR
        <target_names_col> IN(<source_col_list>))
        AS <result_table_alias>
```

כתזכורת, join משלב תת–קבוצה (תלוי בסוג ה–join) של הצעדים הבאים:

1. J1: בצע מכפלה קרטזית בין קלטים שמאל וימין.
2. J2: יישם פסוקית ON.
3. J3: הוסף שורות חיצוניות.

APPLY

אופרטור APPLY משלב תת–קבוצה (תלוי בסוג ה–apply) של שני הצעדים הבאים:

1. A1: יישם ביטוי טבלה ימנית עבור כל שורה מהטבלה השמאלית.
2. A2: הוסף שורות חיצוניות.

אופרטור APPLY למעשה מחיל את ביטוי הטבלה הימנית על כל שורה מקלט שמאל. תוכל לחשוב על כך כדבר דומה ל–join, בהבדל אחד חשוב – ביטוי הטבלה הימנית יכול להתייחס לטורי קלט שמאל כמתאם (קורלציה). כאילו שב–join אין קדימות לאף אחד מהקלטים כאשר הם מוערכים. עבור APPLY, לוגית נוח לדמיין שקלט שמאל מוערך ראשון, ואז קלט ימין מוערך פעם אחת לכל שורה מקלט משמאל.

צעד A1 מיושם הן ב– CROSS APPLY והן ב– OUTER APPLY. צעד A2 מיושם אך ורק עבור OUTER APPLY. CROSS APPLY אינו מחזיר שורה חיצונית (שמאלית) אם ביטוי הטבלה הפנימי (ימני) מחזיר סט ריק עבורה. OUTER APPLY יחזיר שורה כזו, עם NULLs במאפייני ביטויי הטבלה הפנימיים.

למשל, השאילתה הבאה מחזירה את שתי ההזמנות האחרונות (בהנחה, לצורך דוגמה זו, ש–orderid מייצג סדר כרונולוגי) עבור כל לקוח, ומייצרת את הפלט המוצג בטבלה 13–1:

```
SELECT C.customerid, city, orderid
FROM dbo.Customers AS C
  CROSS APPLY
    (SELECT TOP(2) orderid, customerid
     FROM dbo.Orders AS O
     WHERE O.customerid = C.customerid
     ORDER BY orderid DESC) AS CA;
```

טבלה 13–1: שתי הזמנות אחרונות של כל לקוח

customerid	*city*	*orderid*
FRNDO	Madrid	2
FRNDO	Madrid	1
KRLOS	Madrid	5
KRLOS	Madrid	4
MRPHS	Zion	6

שים לב שלקוח FISSA אינו מופיע בפלט מכיוון שביטוי הטבלה CA החזיר קבוצה ריקה עבור שורת הלקוח. אם ברצונך להחזיר גם לקוחות שלא ביצעו הזמנות, עליך להשתמש ב– OUTER APPLY כלהלן, ותקבל את הפלט המוצג בטבלה 14–1:

```
SELECT C.customerid, city, orderid
FROM dbo.Customers AS C
  OUTER APPLY
    (SELECT TOP(2) orderid, customerid
     FROM dbo.Orders AS O
     WHERE O.customerid = C.customerid
     ORDER BY orderid DESC) AS OA;
```

טבלה 14–1: שתי הזמנות אחרונות לכל לקוח, כולל לקוחות שלא ביצעו הזמנות

customerid	*city*	*orderid*
FISSA	Madrid	NULL
FRNDO	Madrid	2

customerid	city	orderid
FRNDO	Madrid	1
KRLOS	Madrid	5
KRLOS	Madrid	4
MRPHS	Zion	6

מידע נוסף: לפרטים נוספים על אופרטור APPLY, אנא עיין בפרק 7.

PIVOT

האופרטור PIVOT מאפשר לך בעיקרון לסובב על ציר (pivot) נתונים, ממצב של קבוצות מרובות שורות, למצב של ריבוי טורים בשורה יחידה עבור כל קבוצה, תוך כדי ביצוע חישובי צבירה כחלק מהתהליך.

בטרם אסביר ואדגים את הצעדים הלוגיים המעורבים בשימוש באופרטור PIVOT, בחן את השאילתה הבאה, בה אשתמש מאוחר יותר כקלט שמאל עבור האופרטור PIVOT:

```
SELECT C.customerid, city,
  CASE
    WHEN COUNT(orderid)  = 0 THEN 'no_orders'
    WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
    WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
  END AS category
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.customerid = O.customerid
GROUP BY C.customerid, city;
```

שאילתה זו מחזירה קטגוריות של לקוח המבוססות על כמות ההזמנות (ללא הזמנות, עד שתי הזמנות, למעלה משתי הזמנות), ומפיקה את התוצאה המוצגת בטבלה 15–1.

טבלה 15–1: קטגוריות לקוח בהתאם לכמות הזמנות

customerid	city	category
FISSA	Madrid	no_orders
FRNDO	Madrid	upto_two_orders
KRLOS	Madrid	more_than_two_orders
MRPHS	Zion	upto_two_orders

נניח שברצונך לדעת את מספר הלקוחות בכל קטגוריה בכל עיר. שאילתת ה–PIVOT הבאה מאפשרת לך לקבל מידע זה, ומפיקה את הפלט המופיע בטבלה 16–1:

```
SELECT city, no_orders, upto_two_orders, more_than_two_orders
FROM (SELECT C.customerid, city,
        CASE
          WHEN COUNT(orderid)  = 0 THEN 'no_orders'
          WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
          WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
        END AS category
      FROM dbo.Customers AS C
        LEFT OUTER JOIN dbo.Orders AS O
          ON C.customerid = O.customerid
      GROUP BY C.customerid, city) AS D
  PIVOT(COUNT(customerid) FOR
     category IN([no_orders],
                 [upto_two_orders],
                 [more_than_two_orders])) AS P;
```

טבלה 16–1: מספר לקוחות בכל קטגוריה בכל עיר

city	*no_orders*	*upto_two_orders*	*more_than_two_orders*
Madrid	1	1	1
Zion	0	1	0

אל תיתן לשאילתה המייצרת את הטבלה הנגזרת D לבלבל אותך. ככל הנוגע לך, האופרטור PIVOT מקבל כקלט שמאל שלו ביטוי טבלה שנקרא D, המכיל את קטגוריות הלקוח.

אופרטור PIVOT כולל את שלושת השלבים הלוגיים להלן:

1. P1: קיבוץ מרומז.
2. P2: בידוד ערכים.
3. P3: יישום פונקציית הצבירה.

השלב הראשון (P1) מאוד קשה לתפיסה. ניתן לראות בשאילתה שאופרטור PIVOT מתייחס לשניים מהטורים ב–D כקלטים (customerid ו–category). השלב הראשון מקבץ בעקיפין את השורות מ–D בהתבסס על כל הטורים שלא הוזכרו בקלטים של PIVOT, כאילו היה חבוי שם GROUP BY. במקרה שלנו, רק טור העיר לא הוזכר כאף אחד מהקלטים של PIVOT. כך מתקבלת קבוצה לכל עיר (Madrid ו–Zion במקרה שלנו).

שים לב: שלב הקיבוץ המרומז של PIVOT אינו מחליף פסוקית GROUP BY מפורשת, במידה שכזו מופיעה בשאילתה. PIVOT בסופו של דבר יפיק טבלת תוצאה וירטואלית, שבתורה תהווה קלט לשלב הלוגי הבא, בין אם תהיה זו פעולה טבלאית אחרת או שלב ה–WHERE. כפי שתיארתי מוקדם יותר בפרק, לאחר שלב WHERE עשוי לבוא שלב GROUP BY. כך שכאשר הן PIVOT והן GROUP BY מופיעים בשאילתה, מתקבלים שני שלבי קיבוץ נפרדים – אחד כשלב הראשון של PIVOT (P1) ואחד מאוחר יותר כשלב GROUP BY של השאילתה.

השלב השני של PIVOT (P2) מבודד ערכים מקבילים לטורי יעד. לוגית, הוא משתמש בביטוי CASE הבא עבור כל טור יעד המצוין בפסוקית IN:

```
CASE WHEN <source_col> = <target_col_element> THEN <expression> END
```

במצב זה, מיושמים לוגית שלושת הביטויים הבאים:

```
CASE WHEN category = 'no_orders'            THEN customerid END,
CASE WHEN category = 'upto_two_orders'      THEN customerid END,
CASE WHEN category = 'more_than_two_orders' THEN customerid END
```

שים לב: לביטוי CASE ללא פסוקית ELSE קיים ELSE NULL מרומז.

ביטוי CASE יחזיר את קוד הלקוח לכל טור יעד, רק אם לשורת המקור הייתה את הקטגוריה המקבילה; אחרת CASE יחזיר NULL.

השלב השלישי של PIVOT (P3) מיישם את פונקציית הצבירה המוגדרת על כל ביטוי CASE, ומייצר את טורי התוצאה. במקרה שלנו, הביטויים הופכים לוגית ל–

```
COUNT(CASE WHEN category = 'no_orders'
             THEN customerid END) AS [no_orders],
COUNT(CASE WHEN category = 'upto_two_orders'
             THEN customerid END) AS [upto_two_orders],
COUNT(CASE WHEN category = 'more_than_two_orders'
             THEN customerid END) AS [more_than_two_orders]
```

לסיכום, שאילתת ה–PIVOT הקודמת זהה לוגית לשאילתה הבאה:

```
SELECT city,
  COUNT(CASE WHEN category = 'no_orders'
               THEN customerid END) AS [no_orders],
  COUNT(CASE WHEN category = 'upto_two_orders'
               THEN customerid END) AS [upto_two_orders],
  COUNT(CASE WHEN category = 'more_than_two_orders'
               THEN customerid END) AS [more_than_two_orders]
FROM (SELECT C.customerid, city,
        CASE
          WHEN COUNT(orderid)  = 0 THEN 'no_orders'
          WHEN COUNT(orderid) <= 2 THEN 'upto_two_orders'
          WHEN COUNT(orderid)  > 2 THEN 'more_than_two_orders'
        END AS category
      FROM dbo.Customers AS C
        LEFT OUTER JOIN dbo.Orders AS O
          ON C.customerid = O.customerid
      GROUP BY C.customerid, city) AS D
GROUP BY city;
```

מידע נוסף: לפרטים נוספים על האופרטור PIVOT, אנא עיין בפרק 6.

UNPIVOT

UNPIVOT היא הפעולה ההפוכה ל–PIVOT, סיבוב נתונים על ציר ממצב של ערכי טבלה מרובים באותה שורה לשורות מרובות, כל אחת עם ערך טור מקור שונה.

לפני שאדגים את השלבים הלוגיים של UNPIVOT, ראשית הרץ את הקוד בקטע–קוד 4–1, ליצירה ומילוי של הטבלה PivotedCategories.

קטע–קוד 4–1: יצירה ומילוי של הטבלה PivotedCategories

```
SELECT city, no_orders, upto_two_orders, more_than_two_orders
INTO dbo.PivotedCategories
FROM (SELECT C.customerid, city,
        CASE
          WHEN COUNT(orderid)  =  0 THEN 'no_orders'
          WHEN COUNT(orderid) <=  2 THEN 'upto_two_orders'
          WHEN COUNT(orderid)  >  2 THEN 'more_than_two_orders'
        END AS category
      FROM dbo.Customers AS C
```

```
        LEFT OUTER JOIN dbo.Orders AS O
          ON C.customerid = O.customerid
      GROUP BY C.customerid, city) AS D
  PIVOT(COUNT(customerid) FOR
    category IN([no_orders],
                [upto_two_orders],
                [more_than_two_orders])) AS P;

UPDATE dbo.PivotedCategories
  SET no_orders = NULL, upto_two_orders = 3
WHERE city = 'Madrid';
```

לאחר שתריץ את הקוד בקטע-קוד 1-4, הטבלה PivotedCategories תכיל את הנתונים המוצגים בטבלה 1-17.

טבלה 1-17: תוכן של טבלת PivotedCategories

city	*no_orders*	*upto_two_orders*	*more_than_two_orders*
Madrid	NULL	3	1
Zion	0	1	0

אשתמש בשאילתה הבאה כדוגמה כדי לתאר את שלבי העיבוד הלוגי המעורבים באופרטור UNPIVOT:

```
SELECT city, category, num_custs
FROM dbo.PivotedCategories
  UNPIVOT(num_custs FOR
    category IN([no_orders],
                [upto_two_orders],
                [more_than_two_orders])) AS U
```

שאילתה זו מבצעת unpivot (או מחלקת) את קטגוריות הלקוח מכל שורת מקור לשורה נפרדת לכל קטגוריה, ומפיקה את הפלט המוצג בטבלה 1-18.

טבלה 1-18: קטגוריות לקוח לאחר unpivot

city	*category*	*num_custs*
Madrid	upto_two_orders	3

city	*category*	*num_custs*
Madrid	more_than_two_orders	1
Zion	no_orders	0
Zion	upto_two_orders	1
Zion	more_than_two_orders	0

שלושת שלבי העיבוד הלוגי להלן מעורבים בפעולת UNPIVOT:

1. U1: שכפול שורות.
2. U2: בידוד ערכי טורי יעד.
3. U3: סינון החוצה של שורות עם NULL.

הצעד הראשון (U1) משכפל שורות מביטוי הטבלה השמאלי המסופק ל–UNPIVOT כקלט (במקרה שלנו, PivotedCategories). כל שורה משוכפלת פעם אחת לכל טור מקור המופיע בפסוקית IN. מכיוון שישנם שלושה שמות טורים בפסוקית IN, כל שורת מקור תשוכפל שלוש פעמים. טבלת התוצאה הווירטואלית תכלול טור חדש המכיל את שמות טורי המקור כמחרוזות תווים. טור זה יקבל את השם המוגדר מייד לפני פסוקית IN (במקרה שלנו, קטגוריה). הטבלה הווירטואלית המוחזרת מהשלב הראשון בדוגמה שלנו מוצגת בטבלה 19–1.

טבלה 19–1: טבלה וירטואלית המוחזרת מהשלב הראשון של UNPIVOT

city	*no_ orders*	*upto_two_ orders*	*more_than_two_ orders*	*category*
Madrid	NULL	3	1	no_orders
Madrid	NULL	3	1	upto_two_orders
Madrid	NULL	3	1	more_than_two_orders
Zion	0	1	0	no_orders
Zion	0	1	0	upto_two_orders
Zion	0	1	0	more_than_two_orders

השלב השני (U2) מבודד את ערכי טורי היעד. טור היעד שיכיל את הערכים, יקבל את השם המוגדר מייד לפני פסוקית FOR (במקרה שלנו, num_custs). שם טור היעד יכיל את הערך מהטור המקביל לקטגוריה של השורה הנוכחית מהטבלה הווירטואלית. הטבלה הווירטואלית המוחזרת מצעד זה בדוגמה שלנו מוצגת בטבלה 20–1.

טבלה 1-20: טבלה וירטואלית המוחזרת מהשלב השני של UNPIVOT

city	*category*	*num_custs*
Madrid	no_orders	NULL
Madrid	upto_two_orders	3
Madrid	more_than_two_orders	1
Zion	no_orders	0
Zion	upto_two_orders	1
Zion	more_than_two_orders	0

השלב השלישי של UNPIVOT (U3) מסנן החוצה שורות עם NULL בטור ערך התוצאה (במקרה שלנו, num_custs). הטבלה הווירטואלית המוחזרת מצעד זה בדוגמה שלנו מוצגת בטבלה 1–21.

טבלה 1–21: טבלה וירטואלית מוחזרת מהשלב השלישי של UNPIVOT

city	*category*	*num_custs*
Madrid	upto_two_orders	3
Madrid	more_than_two_orders	1
Zion	no_orders	0
Zion	upto_two_orders	1
Zion	more_than_two_orders	0

כשתסיים להתנסות עם אופרטור UNPIVOT, הסר את טבלה PivotedCategories:

```
DROP TABLE dbo.PivotedCategories;
```

מידע נוסף: לפרטים נוספים על האופרטור UNPIVOT, אנא עיין בפרק 6.

הפסוקית OVER

פסוקית OVER מאפשרת לך לבקש חישובים מבוססי–חלון. ב– SQL Server 2005, פסוקית זו היא אפשרות חדשה לפונקציות צבירה (הן פונקציות צבירה מובנות והן פונקציות צבירה מבוססות CLR [Common Language Runtime]) וזהו אלמנט חובה עבור ארבע פונקציות הדירוג החדשות (ROW_NUMBER, RANK, DENSE_RANK ו–NTILE). כאשר מוגדרת פסוקית OVER, הקלט שלה, במקום רשימת GROUP BY של השאילתה, מציין את חלון השורות עליהן מחושבת פונקציית הצבירה או הדירוג.

בשלב זה לא אדון ביישומים של חישובים מבוססי–חלון, כמו כן לא אתעמק בדרך המדויקת בה פונקציות אלו עובדות; אסביר רק את השלבים בהם פסוקית OVER ניתנת לשימוש. פסוקית OVER תכוסה בפירוט רב יותר בפרקים 4 ו–6.

פסוקית OVER ניתנת לשימוש רק באחד משני שלבים: שלב ה–SELECT (8) ושלב ה– ORDER BY (10). לפסוקית זו יש גישה לטבלה הווירטואלית המסופקת כקלט לשלב זה. קטע–קוד 5–1 מדגיש את שלבי העיבוד הלוגי בהם פסוקית OVER ניתנת לשימוש.

קטע–קוד 5–1: פסוקית OVER בעיבוד לוגי של שאילתה

```
(8)   SELECT (9)  DISTINCT (11)  TOP <select_list>
(1)   FROM <left_table>
(3)     <join_type> JOIN <right_table>
(2)        ON <join_condition>
(4)   WHERE <where_condition>
(5)   GROUP BY <group_by_list>
(6)   WITH {CUBE | ROLLUP}
(7)   HAVING <having_condition>
(10)  ORDER BY <order_by_list>
```

את פסוקית OVER הנך מגדיר לאחר הפונקציה עליה היא מיושמת, ב–select_list או ב–order_by_list.

על אף שלא הסברתי בפירוט כיצד פועלת פסוקית OVER, אדגים את השימוש בה בשני השלבים בהם היא ניתנת לשימוש. הדוגמה הבאה משתמשת בפסוקית OVER עם פונקציית הצבירה COUNT ברשימת ה–SELECT; הפלט של שאילתה זו מוצג בטבלה 22–1:

```
SELECT orderid, customerid,
  COUNT(*) OVER(PARTITION BY customerid) AS num_orders
FROM dbo.Orders
WHERE customerid IS NOT NULL
  AND orderid % 2 = 1;
```

טבלה 22–1: פסוקית OVER מיושם בשלב ה–SELECT

orderid	*customerid*	*num_orders*
1	FRNDO	1
3	KRLOS	2
5	KRLOS	2

פסוקית PARTITION BY מגדירה את החלון עבור החישוב. הפונקציה COUNT(*) סופרת את מספר השורות בטבלה הווירטואלית המסופקת כקלט לשלב ה–SELECT, בהן customerid שווה לזה בשורה הנוכחית. זכור שהטבלה הווירטואלית המסופקת כקלט לשלב ה–SELECT עברה כבר סינון WHERE – כלומר, קודי לקוח NULL ואפילו קודי הזמנה הוסרו כבר.

ניתן גם להשתמש בפסוקית OVER ברשימת ORDER BY. למשל, השאילתה הבאה ממיינת את השורות לפי המספר הסופי של שורות פלט ללקוח (בסדר יורד), ומייצרת את הפלט המופיע בטבלה 23–1:

```
SELECT orderid, customerid
FROM dbo.Orders
WHERE customerid IS NOT NULL
  AND orderid % 2 = 1
ORDER BY COUNT(*) OVER(PARTITION BY customerid) DESC;
```

טבלה 23–1: פסוקית OVER מיושמת בשלב ORDER BY

orderid	*customerid*
3	KRLOS
5	KRLOS
1	FRNDO

מידע נוסף: לפרטים על השימוש בפסוקית OVER עם פונקציות צבירה (אגרגציה), אנא עיין בפרק 6. לפרטים על השימוש בפסוקית OVER עם פונקציות דירוג, אנא עיין בפרק 4.

פעולות על קבוצות (Set Operations)

SQL Server 2005 תומך בשלוש פעולות על קבוצות: UNION, EXCEPT ו–INTERSECT. UNION היא היחידה הזמינה ב– SQL Server 2000. אופרטורים אלו של SQL מקבילים לאופרטורים המוגדרים בתיאוריה המתמטית של תורת הקבוצות. זהו התחביר עבור שאילתה המיישמת פעולת סט:

```
[(]left_query[)] {UNION [ALL] | EXCEPT | INTERSECT} [(]right_query[)]
[ORDER BY <order_by_list>]
```

פעולות סט משוות בין שורות שלמות משני קלטים. UNION מחזירה תוצאה אחת עם השורות משני הקלטים. אם אפשרות ALL אינה מצוינת, UNION מסירה מהתוצאה שורות כפולות. EXCEPT מחזירה שורות ייחודיות (distinct) המופיעות בקלט שמאל אך לא

בקלט ימין. INTERSECT מחזירה שורות ייחודיות המופיעות בשני הקלטים. ניתן לומר עוד דברים רבים בנוגע לפעולות סט אלו, אך כאן אבחר להתמקד בצעדי העיבוד הלוגי המעורבים בפעולת סט.

פסוקית ORDER BY אינה מותרת בשאילתות הקלט לפעולת הסט. מותר לך להגדיר פסוקית ORDER BY בסוף השאילתה, אך היא תיושם על התוצאה של פעולת הסט.

במושגים של עיבוד לוגי, כל שאילתת קלט מעובדת ראשית בנפרד עם כל השלבים הרלוונטיים לה. אז מיושמת פעולת הסט, ואם מצוינת פסוקית ORDER BY, היא מיושמת על התוצאה.

התבונן לדוגמה בשאילתה הבאה המייצרת את הפלט המופיע בטבלה 24–1:

```
SELECT 'O' AS letter, customerid, orderid FROM dbo.Orders
WHERE customerid LIKE '%O%'

UNION ALL

SELECT 'S' AS letter, customerid, orderid FROM dbo.Orders
WHERE customerid LIKE '%S%'

ORDER BY letter, customerid, orderid;
```

טבלה 24–1: תוצאה של פעולת הקבוצה UNION ALL

letter	*customerid*	*orderid*
O	FRNDO	1
O	FRNDO	2
O	KRLOS	3
O	KRLOS	4
O	KRLOS	5
S	KRLOS	3
S	KRLOS	4
S	KRLOS	5
S	MRPHS	6

ראשית, כל שאילתת קלט מעובדת בנפרד על כל שלבי העיבוד הלוגי הרלוונטיים. השאילתה הראשונה מחזירה טבלה עם הזמנות שבוצעו על ידי לקוחות המכילים את האות **O**. השאילתה השנייה מחזירה טבלה עם הזמנות שבוצעו על ידי לקוחות המכילים את האות **S**. פעולת הסט UNION ALL מצרפת את שני הסטים לאחד. לבסוף, פסוקית ORDER BY ממיינת את השורות לפי letter, customerid ו–orderid.

כדוגמה נוספת לשלבי עיבוד לוגי של פעולות סט, השאילתה הבאה מחזירה לקוחות שלא ביצעו הזמנות:

```
SELECT customerid FROM dbo.Customers
EXCEPT
SELECT customerid FROM dbo.Orders;
```

השאילתה הראשונה מחזירה את סט קודי הלקוח מטבלת Customers ({FISSA, FRNDO, KRLOS, MRPHS}), והשאילתה השנייה מחזירה את סט קודי הלקוח מטבלת Orders ({FRNDO, FRNDO, KRLOS, KRLOS, KRLOS, MRPHS, NULL}). פעולת הסט מחזירה ({FISSA}), שמייצג את השורות מהסט הראשון שלא מופיעות בסט השני. לבסוף, פעולת הסט מסירה שורות כפולות מהתוצאה. במקרה זה אין שורות כפולות להסרה.

שמות טורי התוצאה נקבעים על ידי קלט שמאל של פעולת הסט. טורים במיקומים מקבילים חייבים להיות תואמים בטיפוס הנתונים (datatype) שלהם או לעבור המרה בעקיפין. לסיום, היבט מעניין של פעולות סט הוא שהן מתייחסות ל–NULLs כשווים.

מידע נוסף: תוכל למצוא דיון מעמיק יותר בפעולות סט בפרק 5.

סיכום

הבנת שלבי עיבוד לוגי של שאילתה וההיבטים הייחודיים של SQL חיונית לפיתוח סגנון החשיבה המיוחד הנדרש לצורך תכנות בשפת SQL. על ידי היכרות עם היבטים אלו של השפה, תוכל לייצר פתרונות יעילים ולהסביר את הבחירות שאתה עושה. זכור, הרעיון הוא לשלוט בעקרונות ובטכניקות היסוד.

2

עיבוד פיסי של שאילתות

– מאת לובור קולר

בעוד שהפרק הקודם תיאר את **התוצאה** שהרצת שאילתה צריכה לייצר, פרק זה יסביר **כיצד** Microsoft SQL Server 2005 משיג תוצאה זו.

מרבית מומחי מסדי הנתונים משתמשים בשפת ה–SQL, וכל מוצרי מסדי הנתונים הרלציוניים מכילים ניב כלשהו של סטנדרט SQL. יחד עם זאת, לכל מוצר יש מנגנון עיבוד שאילתות ייחודי משל עצמו. הבנת הדרך בה מנוע מסד הנתונים מעבד שאילתות, מסייעת לארכיטקטים, מעצבים ותוכניתנים לבצע בחירות טובות כאשר הם מעצבים סכמות של מסדי נתונים וכותבים שאילתות.

כאשר שאילתה מגיעה למנוע מסד הנתונים, SQL Server מבצע שני צעדים עיקריים להפקת תוצאת השאילתה הרצויה. הצעד הראשון הוא קומפילציה של השאילתה, המייצרת **תוכנית שאילתה**, והצעד השני הוא הפעלה של תוכנית השאילתה.

קומפילציה של שאילתה ב– SQL Server 2005 מורכבת משלושה צעדים: פירוק (parsing), algebrization (מונח זה יוסבר בהמשך) ואופטימיזציה של שאילתה. לאחר ביצוע שלבים אלה, ה–compiler מאחסן את תוכנית השאילתה האופטימלית ב–cache הפרוצדורות, שם מנוע ההפעלה מעתיק את התוכנית למצב שניתן להפעלה ולאחר מכן מפעיל את הצעדים שבתוכנית השאילתה להפקת תוצאת השאילתה. אם אותה שאילתה או פרוצדורה מאוחסנת מופעלת שוב והתוכנית נמצאת ב–cache הפרוצדורות, יש לדלג על שלב הקומפילציה, והשאילתה או הפרוצדורה המאוחסנת ממשיכה ישירות להפעלה תוך שימוש חוזר באותה תוכנית מאוחסנת.

בפרק זה נבחן כיצד ה– query optimizer מייצר את תוכנית השאילתה וכיצד באפשרותך לשים את ידך הן על התוכנית המשוערת והן על התוכנית בפועל המשמשת לעיבוד השאילתה. נתחיל בדוגמה של התוצר הסופי, יחד עם תיאור המסביר כיצד המוצר מבצע את הפונקציה הרצויה – בדרך זו נבין טוב יותר את התהליך של בניית התוצר עצמו. לפיכך, אתחיל בדוגמה של הפעלת שאילתה דומה לזו איתה עבדנו בפרק 1. בהמשך, לאחר שהעקרונות יובנו, אביט מקרוב יותר לתוך תהליך הקומפילציה של השאילתה ואתאר את הצורות השונות של תוכניות השאילתה.

זרימת נתונים במהלך עיבוד שאילתה

אם ברצונך להפוך את הדוגמה הבאה לחוויה מעשית, התחל את SSMS (SQL Server Management Studio). הרץ את השאילתה המוצגת בקטע–קוד 1–2 על מסד הנתונים Northwind, לאחר שלחצת על הסמל Include Actual Execution Plan, כפי שמוצג בתרשים 1–2.

שים לב: מסד הנתונים Northwind אינו מסופק עם SQL Server 2005. תוכל להוריד את גרסת SQL Server 2000 של Northwind (שעובדת גם על SQL Server 2005) באתר Microsoft:
http://www.microsoft.com/technet/prodtechnol/sql/2000/downloads/default.mspx

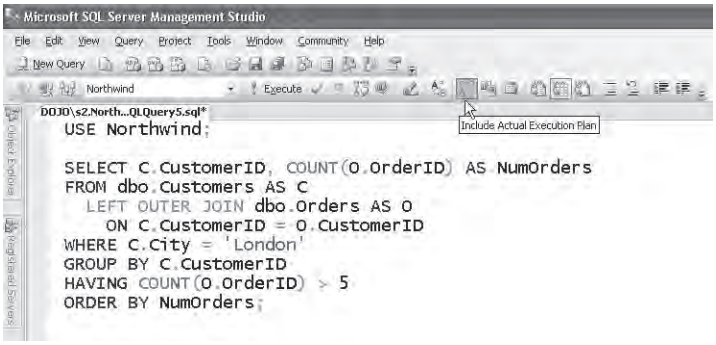
קוד ההתקנה ייצור תיקייה בשם SQL Server 2000 Sample Databases בכונן :C שלך, ושם תמצא instnwnd.sql. הרץ קוד זה ב–SSMS ליצירת מסד הנתונים Northwind. לחלופין, תוכל להשתמש ב– CREATE DATABASE Northwind FOR ATTACH… כדי להצמיד את הקבצים NORTHWND.LDF ו–NORTHWND.MDF למופע שלך של SQL Server 2005.

שים לב: זכור שבאפשרותך להוריד את קוד המקור לספר מאתר www://www.insidetsql.com.

קטע–קוד 1–2: שאילתה על Northwind להדגמת התוצאה של תהליך האופטימיזציה

```
USE Northwind;

SELECT C.CustomerID, COUNT(O.OrderID) AS NumOrders
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID
WHERE C.City = 'London'
GROUP BY C.CustomerID
HAVING COUNT(O.OrderID) > 5
ORDER BY NumOrders;
```

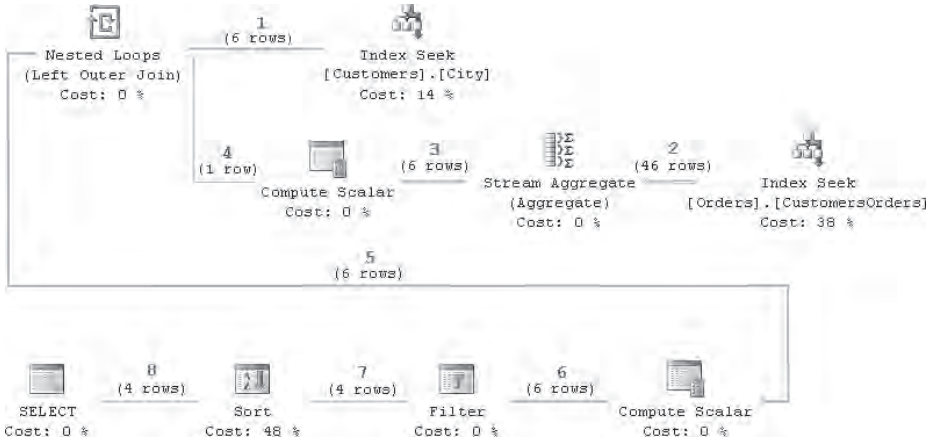
תרשים 2–1: Include Actual Execution Plan



תקבל תוכנית שאילתה גרפית דומה לזו שבתרשים 2–2 בחלונית Execution Plan של ה–SSMS.

שים לב: תוכנית העבודה בתרשים 2–2 מכילה כמה אלמנטים אשר נוספו למטרות הדגמה ולא יופיעו בתוכנית שלך. לכל חץ, הוספתי את מספר השורות המשוער בסוגריים ומספר מזהה בו אשתמש בסעיף הבא.

תרשים 2–2: תוכנית עבודה לשאילתה בקטע–קוד 2–1



שאילתה זו מחזירה את ה–ID (CustomerID) ואת מספר ההזמנות שבוצעו (NumOrders) לכל הלקוחות מלונדון שביצעו למעלה מחמש הזמנות. התוצאה מוצגת בטבלה 1–2.

טבלה 1–2: פלט השאילתה בקטע–קוד 1–2

CustomerID	*NumOrders*
EASTC	8
SEVES	9
BSBEV	10
AROUT	13

כיצד מפעיל SQL Server את התוכנית המוצגת בתרשים 2–2 להפקת התוצאה הרצויה?

ההפעלה של הענפים הבודדים אינה רלוונטית. אדגים את התהליך בדוגמה הבאה, בה SQL Server מעביר את הפעילות שלו בין שני הענפים של האופרטור Nested Loops. לשם התחלה, זכור שהחיצים האפורים בתרשים 2–2 מייצגים זרימת נתונים – שורות הנוצרות על ידי האופרטור, נצרכות על ידי האופרטור הבא בכיוון החיצים. עובי החיצים מתאים למספר היחסי של שורות שה–optimizer מעריך שיזרמו דרך הקישור.

המנוע מתחיל את ההפעלה בביצוע ה– Index Seek שבראש תרשים 2–2 על טבלת Customers – והוא יבחר את השורה הראשונה בה הלקוח הוא מלונדון. אם תרחף עם הסמן מעל הסמל Index Seek מול טבלת Customers, תוכל לראות את ביטוי החיפוש בחלון–צץ תחת Seek Predicates, Prefix: [Northwind].[dbo].[Customers].City = N'London' כפי שמוצג בתרשים 3–2. השורה הנבחרת מועברת לאופרטור Nested Loops על חץ 1, וברגע שהיא מגיעה ל– Nested Loops, מופעל הצד הפנימי של ה– Nested Loops. במקרה שלנו, בתרשים 2–2 הצד הפנימי של האופרטור Nested Loops מורכב מהאופרטורים Compute Scalar, Stream Aggregate ו– Index Seek המחוברים ל– Nested Loops על ידי חיצים 4, 3 ו–2 בהתאמה.

תרשים 3–2: חלון מידע צץ עבור האופרטור Index Seek

Index Seek
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001636
Estimated Operator Cost	0.0032886 (14%)
Estimated Subtree Cost	0.0032886
Estimated Number of Rows	6
Estimated Row Size	17 B
Ordered	True
Node ID	4

Object
[Northwind].[dbo].[Customers].[City] [C]
Output List
[Northwind].[dbo].[Customers].CustomerID
Seek Predicates
Prefix: [Northwind].[dbo].[Customers].City = N'London'

אם נחקור את האופרטור Index Seek בצד הפנימי של ה–Nested Loops בתרשים 2–2, נמצא שביטוי החיפוש שלו הוא:

Prefix: [Northwind].[dbo].[Orders].CustomerID = [Northwind].[dbo].[Customers].[CustomerID] as [C].[CustomerID]

ניתן לראות שהערך C.CustomerID משמש לחיפוש בתוך טבלת Orders להחזרת כל ההזמנות עבור ה–CustomerID. זוהי דוגמה בה הצד הפנימי של ה– Nested Loops מתייחס לערך שהושג בצד האחר, שנקרא הצד החיצוני של ה– Nested Loops.

לאחר שנמצאו כל ההזמנות ללקוח הראשון מלונדון, הן מועברות דרך החץ המסומן ב–2 לאופרטור Stream Aggregate, שם הן נספרות ותוצאת הספירה הנקראת Expr1004, מאוחסנת בשורה על ידי האופרטור Compute Scalar בין חיצים 3 ו–4. בהמשך, השורה המורכבת מה–CustomerID וספירת ההזמנות, מועברת דרך חיצים 5 ו–6 לאופרטור Filter עם הביטוי [Expr1004] > (5). Expr1004 מייצג את הביטוי COUNT(O.OrderID), וה–(5) הוא הקבוע בו השתמשנו בשאילתה להגבלת התוצאה רק ללקוחות עם למעלה מחמש הזמנות.

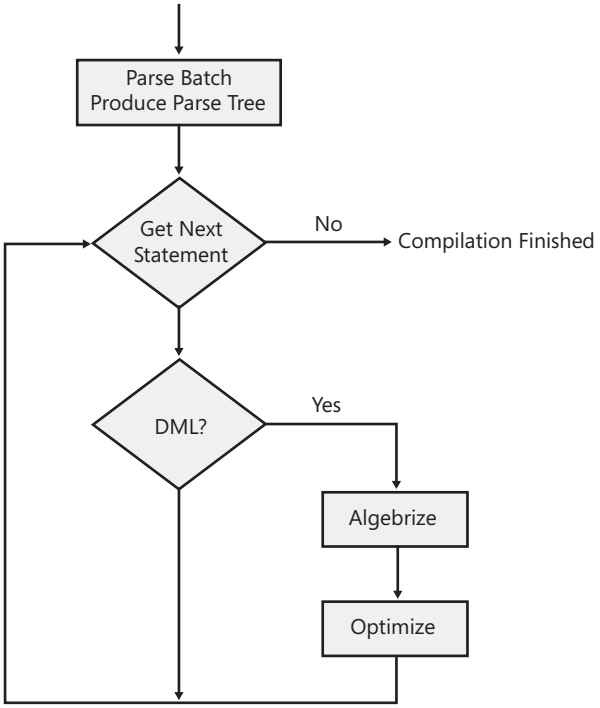
שוב ניתן לראות את הביטוי בחלון–צץ כאשר אתה מציב את הסמן על הסמל Filter. אם הביטוי מחזיר true (כלומר ללקוח יש יותר מחמש הזמנות), השורה מועברת לאופרטור Sort דרך חץ 7. שים לב ש– SQL Server אינו יכול לפלוט שום שורה מה–Sort עד שהוא אוסף את כל השורות שצריכות להיות ממוינות. זאת מכיוון שהשורה האחרונה המגיעה ל–Sort עשויה להיות זו שצריכה להיות "ראשונה" בסדר הנתון (הלקוח עם המספר הנמוך ביותר של הזמנות שהוא מעל חמש במקרה שלנו). לפיכך, השורות "מחכות" ב–Sort, והתהליך המתואר לעיל חוזר על עצמו עבור הלקוח הלונדוני הבא שיימצא על ידי אופרטור ה– Index Seek על טבלת Customers. ברגע שכל השורות שצריכות להיות מוחזרות מגיעות לאופרטור Sort, הוא יחזיר אותן בסדר הנכון (חץ 8).

קומפילציה

batch הוא מצבור של משפט או מספר משפטי Transact-SQL שעוברים קומפילציה כיחידה אחת. פרוצדורה מאוחסנת היא דוגמה ל–batch. דוגמה נוספת היא סט של משפטים בחלון SQL Query ב–SSMS. הפקודה GO מחלקת סטים של משפטים ל–batches נפרדים. שים לב ש–GO אינו משפט T-SQL. SQLCMD, OSQL ו–SSMS משתמשים במילה השמורה GO לסימון סוף ה–batch.

SQL Server עושה קומפילציה למשפטי batch לתוך יחידה אחת הניתנת להפעלה הנקראת **תוכנית עבודה (execution plan)**. במהלך הקומפילציה, ה–compiler מרחיב את המשפטים על ידי הוספת האילוצים, הטריגרים ופעולות ה–cascade הרלוונטיים שצריכים להיות מופעלים בעת הרצת המשפט. אם ה–batch שעובר קומפילציה מכיל קריאות לפרוצדורות מאוחסנות או לפונקציות אחרות והתוכניות שלהן אינן ב–cache, הפרוצדורות המאוחסנות והפונקציות עוברות קומפילציה גם הן בצורה רקורסיבית. הצעדים העיקריים בקומפילציה של batch מוצגים בתרשים 4–2.

תרשים 4–2: קומפילציה



חשוב לזכור שקומפילציה והפעלה הם שלבים נפרדים בעיבוד שאילתה ושהפער בין הזמן בו SQL Server עושה קומפילציה לשאילתה לבין הזמן בו השאילתה מופעלת יכול להיות קצר כמיליוניות שנייה בודדות או ארוך כמספר ימים. לשאילתה אד–הוק לרוב אין תוכנית עבודה ב–cache כאשר היא מעובדת; לפיכך, היא עוברת קומפילציה והתוכנית שלה מופעלת מיידית. מצד שני, לאחר שהתכנית עוברת קומפילציה לפרוצדורה מאוחסנת המופעלת לעיתים קרובות, היא עשויה להיות מאוחסנת ב–cache הפרוצדורות למשך זמן ארוך מאוד. זאת מכיוון ש– SQL Server מסיר את התוכניות שאינן בשימוש תכוף מ–cache הפרוצדורות קודם לכן, במידה שהזיכרון נדרש למטרות אחרות, כמו אחסנת תוכניות שאילתה חדשות.

ה–optimizer לוקח בחשבון כמה מעבדים זמינים ל– SQL Server ואת כמות הזיכרון הזמינה להפעלת שאילתות. עם זאת, מספר המעבדים הזמינים להפעלת השאילתה והכמות של הזיכרון הזמין יכולים להשתנות משמעותית מרגע לרגע. חשוב להתייחס לקומפילציה ולהפעלה כשתי פעילויות נפרדות, גם כאשר אתה שולח משפט SQL אד–הוק דרך SSMS ומפעיל אותו מיידית.

כאשר SQL Server מוכן לעבד batch, תוכנית עבודה ל–batch עשויה כבר להיות זמינה ב–cache של SQL Server. אם לא, ה–compiler מבצע קומפילציה ל–batch ומייצר תוכנית שאילתה. תהליך הקומפילציה כולל מספר דברים. תחילה, SQL Server עובר דרך השלבים של פירוק (parsing) וכריכה (binding). פירוק הוא התהליך של בדיקת התחביר והפיכת ה– SQL batch שלך לעץ פירוק (parse tree). פירוק הוא פעולה כללית המשמשת קומפיילרים כמעט בכל שפות התכנות. הדבר היחיד הייחודי ל–parser של SQL Server הוא החוקים בהם הוא משתמש להגדרת תחביר T-SQL תקני.

פירוק בודק למשל, האם טבלה או שם טור מתחילים בספרה. אם נמצאת ספרה, ה–parser מרים דגל שגיאה. עם זאת, פירוק אינו בודק האם טור הנמצא בשימוש בפסוקית WHERE אכן קיים באחת הטבלאות המופיעות בפסוקית FROM; נושא זה מטופל במהלך הכריכה.

תהליך הכריכה קובע את מאפייני האובייקטים אליהם אתה מתייחס במשפטי ה–SQL שלך, והוא בודק האם הסמנטיקה בה אתה משתמש הגיונית. למשל, בעוד ששאילתה הכוללת FROM A JOIN B עשויה לעבור פירוק בהצלחה, כריכה תכשל אם A הוא טבלה ו–B הוא פרוצדורה מאוחסנת.

השלב האחרון בקומפילציה היא אופטימיזציה. על ה–optimizer לתרגם את הבקשה הלא–פרוצדורלית של משפט SQL מבוסס–סטים לפרוצדורה שיכולה להיות מופעלת בהצלחה ולהחזיר את התוצאה הרצויה. בדומה לכריכה, אופטימיזציה מתבצעת על כל משפט בנפרד עבור כל המשפטים ב–batch. לאחר שהקומפיילר מייצר את התוכנית עבור ה–batch ומאחסן את התוכנית ב–cache הפרוצדורות, מופעל עותק מיוחד של הקשר ההפעלה (**execution context**) של התוכנית. SQL Server מאחסן ב–cache את הקשר ההפעלה בדומה לדרך בה הוא מאחסן את תוכניות השאילתה, ואם אותו batch מתחיל

את ההפעלה השנייה בטרם הראשונה מסתיימת, SQL Server ייצר את הקשר ההפעלה השני מאותה תוכנית. תוכל ללמוד עוד על cache הפרוצדורות של SQL Server מהספר: Inside Microsoft SQL Server 2005: Query Processing and Optimization (Microsoft Press, 2006) מאת קיילן דלייני (Kalen Delaney) או מה–whitepaper: Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005 אותו ניתן למצוא ב–: http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx#.

SQL Server אינו מבצע אופטימיזציה לכל משפט ב–batch. הוא מבצע אופטימיזציה רק לסוגי משפטים מסוימים: אלו הניגשים לטבלאות מסד הנתונים ואשר עשויים להיות להם מספר אפשרויות הפעלה. SQL Server מבצע אופטימיזציה לכל משפטי ה–DML (שפה למניפולציית נתונים) – אלו הם משפטי SELECT, INSERT, DELETE ו–UPDATE. בנוסף ל–DML, מבוצעת אופטימיזציה גם לכמה משפטי T-SQL אחרים, CREATE INDEX הוא אחד מהם. רק המשפטים שעברו אופטימיזציה יפיקו תוכניות שאילתה. הדוגמה הבאה מראה שה–optimizer יוצר תוכנית עבור CREATE INDEX:

```
CREATE TABLE dbo.T(a INT, b INT, c INT, d INT);
INSERT INTO dbo.T VALUES(1, 1, 1, 1);
SET STATISTICS PROFILE ON; -- forces producing showplan from execution
CREATE INDEX i ON dbo.T(a, b) INCLUDE(c, d);
SET STATISTICS PROFILE OFF; -- reverse showplan setting
DROP TABLE dbo.T; -- remove the table
```

הוא יפיק את תוכנית השאילתה הבאה שעברה אופטימיזציה:

```
insert [dbo].[T] select *, %%bmk%% from [dbo].[T]
 |--Index Insert(OBJECT:([db].[dbo].[T].[i]))
  |--Sort(ORDER BY:([db].[dbo].[T].[a] ASC, [db].[dbo].[T].[b] ASC, [Bmk1000] ASC))
    |--Table Scan(OBJECT:([db].[dbo].[T]))
```

בדומה למשפט CREATE INDEX, גם המשפטים CREATE STATISTICS, UPDATE STATISTICS, וכמה צורות של ALTER INDEX עוברים אופטימיזציה. מספר משפטים המופעלים פנימית כדי לבצע בדיקת מסד נתונים (DBCC CHECKDB) עוברים אופטימיזציה גם הם. עם זאת, יש לזכור שמתוך המשפטים שעברו אופטימיזציה שאינם משפטי DML, רק CREATE INDEX מייצר showplan על ידי אפשרות ה– STATISTICS PROFILE ואף אחד מהם אינו מייצר תוכנית שאילתה ישירות ב–SSMS. (showplan יוסבר מאוחר יותר בסעיף "עבודה עם תוכנית השאילתה").

Algebrizer[1]

ה–algebrizer הוא רכיב חדש ב– SQL Server, אשר הפונקציה החשובה ביותר שלו היא כריכה (binding). (שים לב שלעיתים כל התהליך המבוצע על ידי ה–algebrizer נקרא **כריכה**, מכיוון שהכריכה היא הפונקציה המשמעותית ביותר של ה–algebrizer). ה–algebrizer מחליף את ה–normalizer של SQL Sever 2000. ה–algebrizer הוא דוגמה טובה למיקוד ארוך הטווח של צוות הפיתוח של SQL Server. עם כל גרסה חדשה של SQL Server, מספר חלקים במוצר מעוצבים ונכתבים מחדש לחלוטין לצורך תחזוקת בסיס קוד בריא. במקרה של ה–algebrizer, העיצוב מחדש נפרש על פני שתי גרסאות של SQL Server. המטרות של צוות הפיתוח היו לא רק לכתוב מחדש, אלא גם לעצב מחדש לחלוטין את הלוגיקה, כדי לשרת הרחבות פונקציונליות נוכחיות ועתידיות של SQL Server.

הפלט של הפירוק – עץ פירוק – הוא הקלט של ה–algebrizer. לאחר ביצוע מספר מעברים דרך עץ הפירוק, ה–algebrizer מפיק את הפלט שלו – הנקרא **עץ עיבוד שאילתה** (query processor tree) – המוכן לאופטימיזציה של השאילתה.

בנוסף לכריכה, העוסקת בעיקר בפענוח שמות (name resolution) על ידי גישה למידע הקטלוג, ה–algebrizer משטח מספר אופרטורים בינאריים ומבצע גזירת טיפוס נתונים (type derivation). בנוסף לבצוע פענוח שמות, ה–algebrizer מבצע כריכה מיוחדת לצבירות ולקיבוצים.

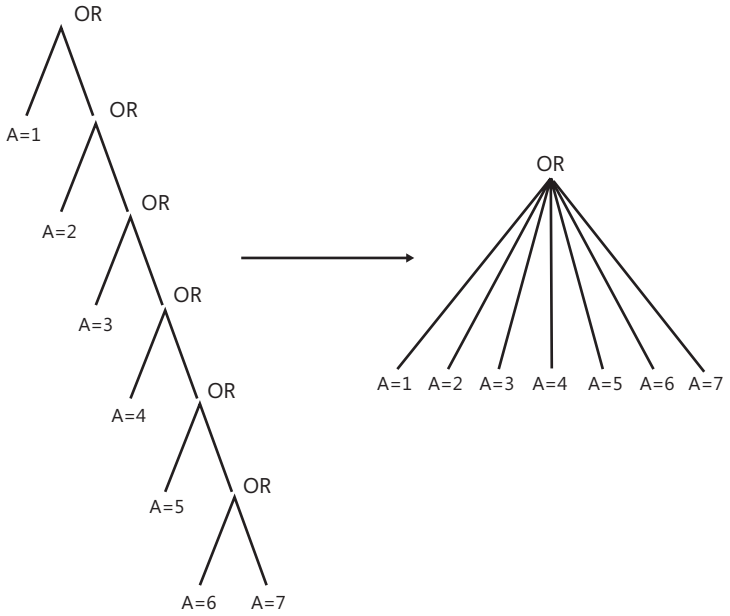
שיטוח אופרטורים

ה–algebrizer מבצע שיטוח של האופרטורים הבינאריים UNION, AND ו–OR. ה–parser תופס אופרטורים אלו כבינאריים בלבד, כפי שמודגם בצד השמאלי של השרטוט בתרשים 2–5 עבור הביטוי (A=1) OR (A=2) OR (A=3) OR (A=4) OR (A=5) OR (A=6) OR (A=7). מצד שני, כל שלבי הקומפילציה לאחר ה–parser מעדיפים להרכיב בכל הזדמנות מספר אופרטורים בינאריים לתוך אופרטור מרובה (n-ary), כפי שנראה בצד הימני של אותו תרשים. דבר זה חשוב במיוחד לרשימות IN ארוכות מאוד שה–parser ממיר לשרשראות של ORs. בין היתר, שיטוח יפתור את מרבית בעיות ה– stack overflow בשלבים הבאים, הנגרמות על ידי עצים עמוקים מאוד. הקוד בתוך SQL Server המבצע את השיטוח נכתב גם הוא בתשומת לב לשימוש בחזרות/לולאות (iterations) ולא ברקורסיה ככל שניתן, כך שה–algebrizer עצמו אינו חשוף לאותה בעיה.

---

[1] תודה ליוג'ין זבוקריצקי (Eugene Zabokritski) על הרשות לכלול מידע על הפטנט algebrizer.

תרשים 5–2: שיטוח OR



פענוח שמות

כל שם טבלה וטור בעץ הפירוק משויך להפניה לאובייקט ההגדרה של הטבלה או הטור. שמות המייצגים את אותו אובייקט מקבלים את אותה הפניה. זהו מידע חשוב לצעד הבא – אופטימיזציה של שאילתה. ה–algebrizer בודק שכל שם אובייקט בשאילתה אכן מתייחס לטבלה או טור תקניים הקיימים בקטלוג המערכת ונראים בטווח השאילתה המסוימת. בהמשך ה–algebrizer מקשר את שם האובייקט עם מידע מהקטלוגים.

פענוח שמות עבור views הוא התהליך של החלפת ההפניה ל–view בעץ ה–view (עץ הפירוק של השאילתה המגדירה את ה– view). SQL Server 2005 מעלה את עצי ה–view ל–cache, כך שהפענוח של view הוא מהיר. Views מפוענחים בצורה רקורסיבית במקרה שהם מתייחסים ל–views נוספים.

גזירת טיפוס נתונים (Type Derivation)

מכיוון ש–T-SQL מודפס סטטית, ה–algebrizer קובע את טיפוס הנתונים של כל צומת (node) בעץ הפירוק. ה–algebrizer מבצע זאת מלמטה למעלה, כשהוא מתחיל מהעלים – טורים (שמידע על הטיפוס שלהם נמצא בקטלוג) וקבועים. אז, עבור צומת שאינו עלה, מידע הטיפוס נגזר מהטיפוס של הילדים ומהמאפיינים של הצומת. דוגמה טובה לגזירת טיפוס נתונים היא התהליך של מציאת טיפוס הנתונים הסופי של שאילתת UNION, כאשר טיפוסי נתונים שונים יכולים להופיע במיקומי טור תואמים (לפרטים נוספים ראה את הפרק "Guidelines for Using Union" ב– SQL Server 2005 Books Online).

כריכת צבירות (aggregate binding)

בחן את הדוגמה הבאה בעזרת טבלה T1 עם טורים c1 ו–c2, וטבלה T2 עם טור x מאותו טיפוס נתונים כמו T1.c2:

```
SELECT c1 FROM dbo.T1
GROUP BY c1
HAVING EXISTS
  (SELECT * FROM dbo.T2
   WHERE T2.x > MAX(T1.c2));
```

SQL Server מחשב את צבירת MAX בשאילתה החיצונית SELECT c1 FROM dbo.T1 GROUP BY c1, על אף שתחבירית היא ממוקמת בשאילתה הפנימית. ה–algebrizer מגיע להחלטה בהתבסס על הארגומנט של הצבירה. השורה התחתונה היא שכל צבירה צריכה להיות קשורה לשאילתה המארחת שלה (המקום בו היא מוערכת נכונה) – דבר זה ידוע **ככריכת צבירה** (aggregate binding).

כריכת קיבוץ (Grouping Binding)

זוהי אולי הפעילות הכי פחות מובנת מאליה מכל הפעילויות המבוצעות על ידי ה–algebrizer. הבה נבחן דוגמה בעזרת טבלה T1 עם טורים c1, c2 ו–c3:

```
SELECT c1 + c2, MAX(c3) FROM dbo.T1 GROUP BY c1 + c2;
```

השימוש בביטוי c1 + c2 ברשימת ה–SELECT לגיטימי, למרות שאם היינו שמים רק את c1 ברשימה הזו, השאילתה הייתה שגויה תחבירית. הסיבה לכך היא **שלביטויים מקובצים** (למשל, אלו עם פסוקיות GROUP BY או HAVING מפורשות) יש חוקי תחביר שונים מביטויים שאינם מקובצים. בפרט, לכל הטורים או הביטויים שאינם צבירה ברשימת ה–SELECT של שאילתה עם GROUP BY, חייבת להיות התאמה ברשימת ה– GROUP BY, והתהליך של וידוא ההתאמה ידוע **ככריכת קיבוץ** (grouping binding).

לרוע המזל, פסוקיות מפורשות כמו GROUP BY או HAVING אינן הגורמים היחידים שעשויים להכריח את רשימת ה–SELECT להפוך למקובצת. לפי חוקי SQL, עצם הנוכחות של פונקציית צבירה המתחברת לרשימה מסוימת, הופכת את רשימת ה–SELECT למקובצת, אפילו אם אין לה שום פסוקיות GROUP BY או HAVING. להלן דוגמה פשוטה:

```
SELECT c1, MAX(c2) FROM dbo.T1;
```

מכיוון שזהו SELECT מקובץ (יש צבירת MAX), השימוש בטור c1, שאינו מופיע בפונקציית צבירה, אינו חוקי והשאילתה שגויה.

תפקיד חשוב של ה–algebrizer הוא לזהות כל שגיאה תחבירית במשפט. הדוגמה הבאה מראה שזוהי משימה שאינה טריוויאלית עבור שאילתות מסוימות עם פונקציות צבירה:

```
SELECT c1, (SELECT T2.y FROM dbo.T2 WHERE T2.x = MAX(T1.c2)) FROM dbo.T1;
```

שאילתה זו שגויה מאותה סיבה כמו השאילתה הקודמת, אך ברור שעלינו להשלים את כריכת הצבירה לכל השאילתה רק בשביל לגלות זאת. ה– MAX(T1.c2) בשאילתה הפנימית חייב להיות מוערך בשאילתה החיצונית בדומה מאוד לשאילתה: SELECT c1, MAX(c2) FROM dbo.T1; שזה עתה הוצגה, ולפיכך, השימוש בטור c1 שאינו מופיע בפונקציית צבירה ברשימת ה–SELECT אינו חוקי והשאילתה שגויה.

אופטימיזציה

אחד מהרכיבים החשובים והמורכבים ביותר המעורבים בעיבוד השאילתות שלך הוא ה– query optimizer. תפקידו של ה–optimizer הוא לייצר תוכנית עבודה יעילה לכל שאילתה ב–batch או בפרוצדורה מאוחסנת. התוכנית מכינה רשימה של הצעדים שעל SQL Server לבצע להרצת השאילתה שלך, והיא כוללת מידע כגון באיזה אינדקס או אינדקסים להשתמש כאשר ניגשים לנתונים מכל טבלה בשאילתה.

התוכנית גם כוללת את האסטרטגיה לעיבוד כל פעולת איחוד, צבירה, מיון וכן כל גישה לטבלה מחולקת למחיצות. התוכנית מראה כוונה לביצוע פעולות בצורה מקבילית — כלומר, כאשר זרם השורות המקורי מחולק למחיצות, הוא מחולק שוב, ואז ממוזג חזרה לזרם יחיד.

ה– query optimizer של SQL Server הוא optimizer מבוסס–עלות, המשמעות היא שהוא מנסה למצוא את תוכנית העבודה הזולה ביותר לכל משפט SQL. עלות התוכנית משקפת את הזמן המשוער להשלמת השאילתה. עבור כל שאילתה, ה–optimizer חייב לנתח את התוכניות האפשריות ולבחור את זו בעלת העלות המשוערת הנמוכה ביותר. ישנם משפטים מורכבים להם מיליוני תוכניות עבודה אפשריות. במקרים אלו, ה– query optimizer לא מנתח את כל השילובים האפשריים. במקום זאת, הוא מנסה למצוא תוכנית עבודה בעלת עלות קרובה יחסית למינימום התיאורטי. בהמשך, אסביר כמה דרכים בהן ה–optimizer יכול להפחית את משך הזמן שהוא משקיע באופטימיזציה.

העלות המשוערת הנמוכה ביותר אינה בהכרח עלות המשאבים הנמוכה ביותר; ה– query optimizer בוחר בתוכנית שמחזירה תוצאות למשתמש במהירות הגבוהה ביותר בעלות משאבים סבירה. למשל, עיבוד שאילתה בצורה מקבילית (שימוש במספר מעבדים בו זמנית לאותה שאילתה) לרוב משתמש ביותר משאבים מאשר עיבוד טורי המשתמש במעבד יחיד, אך השאילתה מסתיימת הרבה יותר מהר כאשר היא מעובדת בצורה מקבילית. ה–optimizer יציע תוכנית עבודה מקבילית להחזרת תוצאות, ו– SQL Server ישתמש בתוכנית עבודה מקבילית כזו אם העומס על השרת אינו מושפע בצורה שלילית.

אופטימיזציה עצמה מערבת מספר צעדים. האופטימיזציה של **תוכנית טריוויאלית** היא הצעד הראשון. הרעיון מאחורי תוכנית טריוויאלית הוא שאופטימיזציה מבוססת–עלות יקרה

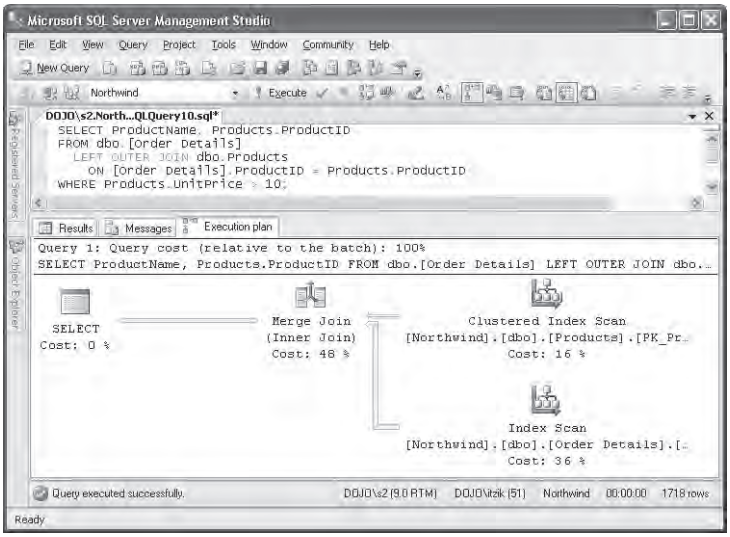
לשימוש ולהרצה. ה–optimizer יכול לנסות הרבה וריאציות אפשריות בחיפוש אחר התוכנית הזולה ביותר. אם SQL Server יודע, על ידי חקירת השאילתה וה–metadata הרלוונטי, שקיימת רק תוכנית מעשית אחת לשאילתה, הוא יכול לחסוך הרבה מהעבודה הנדרשת לאתחול ולביצוע אופטימיזציה מבוססת–עלות. דוגמה נפוצה היא שאילתה המורכבת מ–INSERT עם פסוקית VALUES, כאשר טבלת היעד אינה משתתפת בשום indexed views. במקרה של כזו שאילתה – קיימת רק תוכנית אפשרית אחת. דוגמה אחרת היא SELECT מטבלה יחידה ללא אינדקסים וללא GROUP BY. בשני מקרים אלו, SQL Server צריך רק לייצר את התוכנית ולא לנסות למצוא משהו טוב יותר. התוכנית הטריוויאלית שה–optimizer מוצא היא התוכנית המובנית מאליה, ולרוב היא זולה מאוד. בהמשך הפרק אראה כיצד ניתן לקבוע אם ה–optimizer הפיק תוכנית טריוויאלית לשאילתה מסוימת.

אם ה–optimizer אינו מוצא תוכנית טריוויאלית, SQL Server יבצע מספר הפשטות, שהן לרוב טרנספורמציות תחביריות של השאילתה עצמה, כדי לחפש תכונות ופעולות חלופיות הניתנות לסידור מחדש. SQL Server יכול לבצע פעולות שאינן דורשות התחשבות בעלויות או בניתוח של אילו אינדקסים זמינים, אך מפיקות שאילתה יעילה יותר. דוגמה להפשטה היא להעריך מסנני טבלה יחידה לפני ה–joins. כפי שתואר בפרק 1, המסננים מוערכים לוגית לאחר ה–joins, אך הערכת המסננים לפני ה–joins מפיקה תוצאה נכונה גם כן והיא תמיד יעילה יותר מכיוון שהיא מסירה שורות לא מתאימות לפני פעולת ה–join.

דוגמה נוספת להפשטה היא טרנספורמציה של outer joins ל– inner joins בכמה מקרים, כפי שניתן לראות בתרשים 6–2. ככלל, outer join יוסיף שורות לסט התוצאה של inner join. בשורות נוספות אלו קיים הערך NULL בכל הטורים של הסט הפנימי, אם לא קיימת אף שורה פנימית המספקת את ביטוי ה–join. לפיכך, אם קיים ביטוי על הסט הפנימי שפוסל שורות אלו, התוצאה של ה– outer join זהה לזו של ה– inner join ולעולם לא יהיה זול יותר לייצר את תוצאת ה–outer join. לפיכך, SQL Server משנה את ה– outer join ל– inner join במהלך ההפשטה. בשאילתת ה– OUTER JOIN הבאה, הביטוי Products.UnitPrice > 10 פוסל את כל השורות הנוספות שהופקו על ידי ה– OUTER JOIN, ולפיכך, ה– OUTER JOIN מפושט לכדי INNER join:

```
USE Northwind;
SELECT
  [Order Details].OrderID,
  Products.ProductName,
  [Order Details].Quantity,
  [Order Details].UnitPrice
FROM dbo.[Order Details]
  LEFT OUTER JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 10;
```

תרשים 6–2:
הפשטה של outer join



SQL Server מעלה את המידע הסטטיסטי על האינדקסים והטבלאות, וה–optimizer מתחיל את תהליך האופטימיזציה מבוסס–העלות. ה–optimizer מבוסס–העלות משתמש בסדרה של חוקי טרנספורמציה המנסים מספר חלופות של אסטרטגיות גישה לנתונים, סדר joins, מיקום צבירות, טרנספורמציות של תת–שאילתות, וחוקים אחרים המבטיחים שעדיין תופק התוצאה הנכונה. בדרך כלל, תוצאות נכונות הן אותן תוצאות; עם זאת, חשוב לציין שעבור שאילתות מסוימות קיימת יותר מתוצאה נכונה אחת. למשל, כל סט של 10 הזמנות יהווה תוצאה נכונה עבור השאילתה:

```
SELECT TOP (10) <select_list> FROM Orders;
```

שינוי בנתונים המשפיעים על הניתוח של ה–optimizer מבוסס–העלות עשוי לשנות את התוכנית הנבחרת, שבתורה עשויה לשנות את התוצאה בצורה לא צפויה.

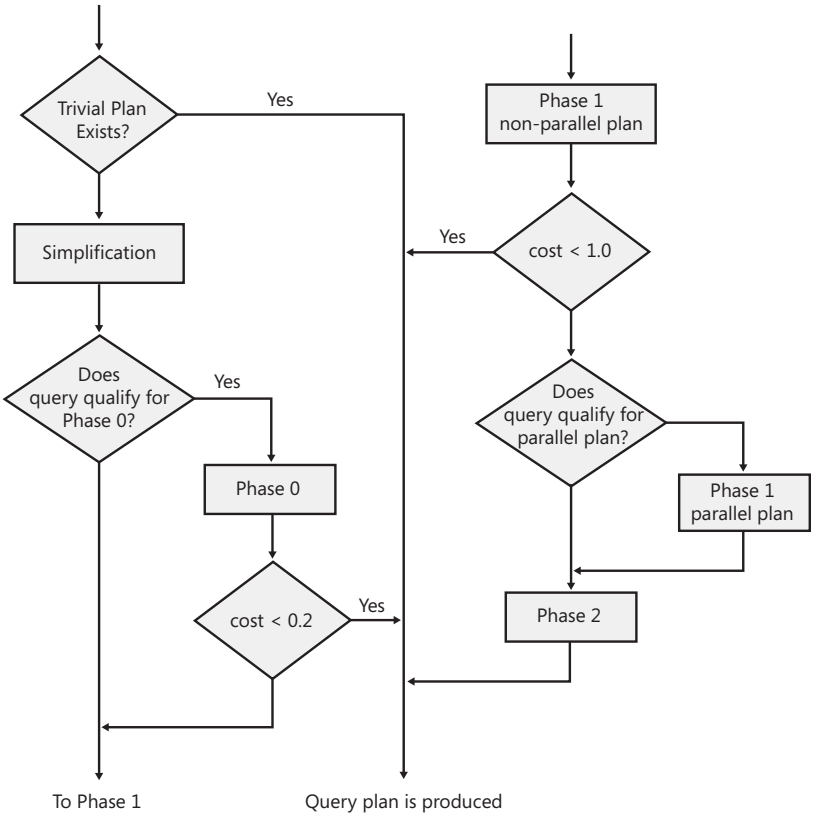
אם ה–optimizer היה משווה את העלות של כל תוכנית תקינה ובוחר בזו שעולה הכי פחות, תהליך האופטימיזציה עשוי היה לארוך זמן רב מאוד – מספר התוכניות התקניות עשוי להיות עצום. לפיכך, האופטימיזציה מחולקת לשלושה **שלבי חיפוש**. לכל שלב מקושרת סדרה של חוקי טרנספורמציה. לאחר כל שלב, SQL Server מעריך את העלות של תוכנית השאילתה הזולה ביותר עד לנקודה זו. אם התוכנית זולה מספיק, SQL Server מסיים את האופטימיזציה ובוחר בתוכנית זו. אם התוכנית אינה זולה מספיק, ה–optimizer מריץ את השלב הבא, המכיל סדרה נוספת של חוקים, לרוב מורכבים יותר.

לשאילתות רבות, על אף שהן מורכבות, יש תוכניות זולות מאוד. אילו SQL Server היה מיישם חוקי טרנספורמציה רבים והיה מנסה סדרי joins שונים, תהליך האופטימיזציה עשוי היה להימשך משמעותית יותר מאשר הפעלת השאילתה עצמה. לפיכך, שלב 0 – השלב הראשון של האופטימיזציה מבוססת–העלות, מכיל סדרה מוגבלת של חוקים ומופעל על שאילתות להן לפחות ארבע טבלאות. מכיוון שהסידור מחדש של joins לבדו מייצר הרבה תוכניות מועמדות אפשריות, ה–optimizer משתמש בכמות מוגבלת של סדרי join בשלב 0,

והוא שוקל רק Hash joins ולולאות מקוננות (Nested Loops). אם שלב זה מוצא תוכנית עם עלות משוערת נמוכה מ–0.2, האופטימיזציה מסתיימת. השאילתות בעלות תוכניות שאילתה סופיות המופקות על ידי שלב 0 נמצאות לרוב ביישומי עיבוד טרנזקציות; לפיכך, שלב זה נקרא גם **שלב עיבוד הטרנזקציות** (Transaction Processing phase).

הצעד הבא, שלב 1 או אופטימיזציית תוכנית מהירה (Quick Plan optimization), משתמש בחוקי טרנספורמציה רבים יותר ומנסה סדרי joins שונים. כאשר השלב מסתיים, אם עלות התוכנית הטובה ביותר היא פחות מ–1.0, האופטימיזציה מסתיימת. עד לנקודה זו, ה–optimizer שוקל רק תוכניות שאילתה שאינן מקביליות. אם יותר ממעבד אחד זמין ל– SQL Server והתוכנית הכי פחות יקרה שנוצרה על ידי שלב 1 יקרה יותר **מסף העלות למקביליות** (אותו עליך לקבוע על ידי שימוש ב–sp_configure כדי למצוא את הערך הנוכחי – ברירת המחדל היא 5), שלב 1 חוזר על עצמו במטרה למצוא את התוכנית המקבילית הטובה ביותר. העלויות של התוכנית הקווית והמקבילית שהושגו בשלב 1 מושוות, ושלב 2, שלב האופטימיזציה המלאה (Full Optimization), מופעל עבור הזולה מבין השתיים. שלב 2 כולל חוקים נוספים – למשל, הוא משתמש בסידור מחדש של Outer Join והחלפה אוטומטית של Indexed Views ל–views מרובי טבלאות. תרשים 7–2 מציג את שלבי האופטימיזציה של שאילתה ב– SQL Server 2005.

תרשים 7–2: שלבי אופטימיזציה של שאילתה



SQL Server 2005 מספק הצצה נרחבת לפעולה שלו עצמו. דבר זה נכון גם לגבי אופטימיזציה של שאילתות. בגרסאות הקודמות של SQL Server, התוצאה היחידה של אופטימיזציה של שאילתה הייתה תוכנית השאילתה; ההיסטוריה של האופטימיזציה הייתה נשכחת, ורק התוצאה הסופית הייתה נשמרת. ב– SQL Server 2005, קיים חלון הצצה חדש לתוך פעילות ה–optimizer – זהו view הניהול הדינמי sys.dm_exec_query_optimizer_info (Dynamic Management View) DMV. View זה מספק מידע מצטבר על כל האופטימיזציות שבוצעו מאז ש– SQL Server אותחל.

בעזרת שימוש ב–DMV זה, תוכל לדעת אילו אירועי optimizer מתרחשים בזמן שה–optimizer מעבד את ה–batches שלך. ה– sys.dm_exec_query_optimizer_info DMV מחזיר שלושה טורים: counter, occurrence ו–value. הטור שנקרא counter מספק את שם אירוע ה–optimizer. הטור occurrence מציג את המספר המצטבר של מופעי אירוע ה–optimizer, וישנם אירועים המשתמשים בטור value כדי לספק ערכים נוספים ספציפיים–לאירוע. למשל, בכל פעם שה–optimizer בוחר בתוכנית טריוויאלית, ערך הטור occurrence של מונה התוכנית הטריוויאלית יגדל באחד. בדומה, תוכל למצוא כמה פעמים כל שלב אופטימיזציה – שלב 0, 1 או 2 – יצא לפועל על ידי חקירת אירועי "search 0", "search 1", או "search 2" התואמים. הטור value משמש, למשל, את האירוע "Tables" – הוא מחזיק את מספר הטבלאות הממוצע אליהן קיימת התייחסות במשפטים עליהם מבוצעת אופטימיזציה. אנא פנה לנושא sys.dm_exec_query_optimizer_info בפרק "SQL Server Language Reference" של Books Online לתיאור מפורט של כל המונים המוחזרים על ידי ה– sys.dm_exec_query_optimizer_info DMV.

כאשר משתמשים ב– sys.dm_exec_query_optimizer_info, יש להיות זהירים עם cache הפרוצדורות. אם ה–cache מכיל כבר את התוכנית עבור השאילתה או ה–batch שלך, שלב האופטימיזציה לא מתקיים ולא יופקו כלל אירועי optimizer. ניתן להשתמש ב– DBCC FREEPROCCACHE לניקוי cache הפרוצדורות כדי להבטיח שהקומפילציה תופעל לאחר מכן. עם זאת עליך להיזהר בשימוש ב– DBCC FREEPROCCACHE בשרתים תפעוליים מכיוון שהוא ימחק את התוכן של cache הפרוצדורות, וכל המשפטים והפרוצדורות השמורות יצטרכו לעבור קומפילציה מחדש.

מכיוון שהמונים של אירועי optimizer מצטברים, תרצה למצוא את הערכים שלהם לפני ואחרי האופטימיזציה של הקוד שלך, אם אתה מעוניין באירועים הנוצרים ובסטטיסטיקות. עם זאת, הפעלה של 'select * from sys.dm_exec_query_optimizer_info' בפני עצמה עשויה לייצר אירועי optimizer. מסיבה זו, בניתי בזהירות את דוגמת הקוד המופיעה בקטע–קוד 2–2, כדי להימנע מקלקול–עצמי של מידע ה–optimizer המסופק על ידי sys.dm_exec_query_optimizer_info ב– SQL Server 2005. המשפט או ה–batch שלך צריכים להיות מוכנסים במקום המסומן בדוגמה בקטע–קוד 2–2. מתוך 38 סוגי האירועים ב–DMV זה, הדוגמה תציג רק את אלו ששינו את הטור occurrence או את הטור value כתוצאה מהמשפט או מה–batch שלך. לאחר שתכניס את הקוד, עליך להריץ את כל ה–batch פעם אחת מ–SSMS כדי לקבל את אירועי ה–optimizer שנוצרו על ידי הקוד שלך.

קטע-קוד 2-2: קוד לקבלת מידע אודות ה-batch שלך מ- sys.dm_exec_query_optimizer_info

```
SET NOCOUNT ON;
USE Northwind; -- use your database name here
DBCC FREEPROCCACHE; -- empty the procedure cache
GO
-- we will use tempdb..OptStats table to capture
-- the information from several executions
-- of sys.dm_exec_query_optimizer_info
IF (OBJECT_ID('tempdb..OptStats') IS NOT NULL)
  DROP TABLE tempdb..OptStats;
GO
-- the purpose of this statement is
-- to create the temporary table tempdb..OptStats
SELECT 0 AS Run, *
INTO tempdb..OptStats
FROM sys.dm_exec_query_optimizer_info;
GO
-- this will populate the procedure cache
-- with this statement's plan so that it will not
-- generate any optimizer events when executed
-- next time
-- the following GO is intentional to ensure
-- the query plan reuse will happen for the following
-- INSERT for its next invocation in this script
GO
INSERT INTO tempdb..OptStats
  SELECT 1 AS Run, *
  FROM sys.dm_exec_query_optimizer_info;
GO
-- same reason as above; observe the "2" replaced "1"
-- therefore, we will have a different plan
GO
INSERT INTO tempdb..OptStats
  SELECT 2 AS Run, *
  FROM sys.dm_exec_query_optimizer_info;
GO
-- empty the temporary table
TRUNCATE TABLE tempdb..OptStats
GO
-- store the "before run" information
-- in the temporary table with the output
```

```
-- of sys.dm_exec_query_optimizer_info
-- with value "1" in the column Run
 GO
 INSERT INTO tempdb..OptStats
   SELECT 1 AS Run, *
   FROM sys.dm_exec_query_optimizer_info;
 GO
-- your statement or batch is executed here
 /*** the following is an example
 SELECT C.CustomerID, COUNT(O.OrderID) AS NumOrders
 FROM dbo.Customers AS C
   LEFT OUTER JOIN dbo.Orders AS O
     ON C.CustomerID = O.CustomerID
 WHERE C.City = 'London'
 GROUP BY C.CustomerID
 HAVING COUNT(O.OrderID) > 5
 ORDER BY NumOrders;
***/
 GO
-- store the "after run" information
-- in the temporary table with the output
-- of sys.dm_exec_query_optimizer_info
-- with value "2" in the column Run
 GO
 INSERT INTO tempdb..OptStats
   SELECT 2 AS Run, *
   FROM sys.dm_exec_query_optimizer_info;
 GO
-- extract all "events" that changed either
-- the Occurrence or Value column value between
-- the Runs 1 and 2 from the temporary table.
-- Display the values of Occurrence and Value
-- for all such events before (Run1Occurrence and
-- Run1Value) and after (Run2Occurrence and
-- Run2Value) executing your batch or query.
-- This is the result set generated by the script.
 WITH X (Run,Counter, Occurrence, Value)
 AS
 (
   SELECT *
   FROM tempdb..OptStats WHERE Run=1
 ),
```

```
Y (Run,Counter, Occurrence, Value)
AS
(
  SELECT *
  FROM tempdb..OptStats
  WHERE Run=2
)
SELECT X.Counter, Y.Occurrence-X.Occurrence AS Occurrence,
  CASE (Y.Occurrence-X.Occurrence)
    WHEN 0 THEN (Y.Value*Y.Occurrence-X.Value*X.Occurrence)
        ELSE (Y.Value*Y.Occurrence-X.Value*X.Occurrence)/
              (Y.Occurrence-X.Occurrence)
  END AS Value
FROM X JOIN Y
  ON (X.Counter=Y.Counter
      AND (X.Occurrence<>Y.Occurrence OR X.Value<>Y.Value));
GO
-- drop the temporary table
DROP TABLE tempdb..OptStats;
GO
```

אם תרצה להשתמש בקוד זה לחקירת אירועי ה–compiler, המונים והערכים התואמים למשפט מקטע–קוד 1–2, עליך לשלב את המשפט שלך בקוד לעיל במקום המסומן על ידי ההערה – "-- your statement or batch is executed here" ולהריץ את הקוד. תוכל לראות את התוצאה של המשפט עצמו ולאחריו את טבלה 2–2. מהמונים, ניתן לראות ש–SQL Server ביצע אופטימיזציה ל–batch המורכב ממשפט יחיד בתוך 0.008752 שניות; שהעלות של התוכנית הסופית היא 0.023881; שה–DOP הוא 0 (כלומר תוכנית סריאלית); שהופעלה אופטימיזציה יחידה; שרק שלב 1 (המקביל ל"חיפוש 1") של אופטימיזציה בוצע והשתמש ב–647 משימות חיפוש בתוך 0.00721 שניות (ה"חיפוש" הוא כמעט תמיד החלק היקר ביותר בקומפילציה של שאילתה), ושלשאילתה יש שתי טבלאות. אם ה–batch מורכב ממספר משפטים, הטור value בטבלה 2–2 יכיל ערכים ממוצעים של המונים והסטטיסטיקות.

טבלה 2–2: מוני אירועי ה–optimizer עבור המשפט מקטע–קוד 1–2

Counter	*Occurrence*	*Value*
Elapsed time	1	0.008752
Final cost	1	0.023881
Maximum DOP	1	0

Counter	*Occurrence*	*Value*
Optimizations	1	1
search 1	1	1
search 1 tasks	1	647
search 1 time	1	0.00721
Tables	1	2
Tasks	1	647

עבודה עם תוכנית שאילתה

Showplan הוא המונח המשמש את משתמשי SQL Server כדי לכנות את הייצוג המילולי, הגרפי או ה–XML של תוכנית השאילתה המיוצרת על ידי ה– query optimizer. אנו משתמשים בו גם כפועל כדי לכנות את התהליך להשגת תוכנית השאילתה. Showplan מציג מידע בנוגע לדרך בה SQL Server יעבד (או עיבד) את השאילתה. showplan מראה האם אינדקסים היו בשימוש או האם סריקת טבלה הכרחית עבור כל טבלה בתוכנית השאילתה. הוא גם מציין את הסדר בו יופעלו הפעולות השונות בתוכנית. קריאת הפלט של showplan היא אומנות כמו שהיא מדע, אך האמת היא שכל הדרוש כדי לנתח אותו הוא הרבה אימון. אני מקווה שהתיאור והדוגמאות בסעיף זה יספיקו כדי לתת לך התחלה טובה.

שים לב: לאורך הספר תמצא דיונים בהם ינותחו תוכניות שאילתה; לפיכך, חלקים נרחבים מהפרק הזה והפרק הבא מתארים כיצד יש לעבוד עם תוכניות שאילתה וכיצד יש לנתח אותן. פרק זה ילמד אותך כיצד לקבל צורות שונות של תוכניות שאילתה, בעוד שפרק 3 ילמד אותך כיצד לבחון אותן מנקודת מבט של כוונון–שאילתה. ייתכן שתמצא חפיפה כלשהי בתוכן של שני הפרקים, אך הדיונים חיוניים לצורך אספקת רקע מקיף להמשך הספר.

SQL Server 2005 יכול לייצר showplans באחד משלושה פורמטים: גרפי, טקסט ו–XML. באשר לתוכן, SQL Server יכול לייצר תוכניות המכילות אופרטורים בלבד, תוכניות המכילות גם עלויות משוערות, ותוכניות המכילות גם מידע זמן–ריצה נוסף. טבלה 3–2 מסכמת את הפקודות ואת הממשקים המשמשים להשגת תוכניות השאילתה עם תוכן שונה בפורמטים השונים:

טבלה 3–2: פקודות המייצרות פורמטים שונים של showplan

Content	Format		
	Text	XML	Graphical
Operators	SET SHOWPLAN_TEXT ON	N/A	N/A
Operators and estimated costs	SET SHOWPLAN_ALL ON	SET SHOWPLAN_XML ON	Display Estimated Execution Plan in Management Studio
Run-time info	SET STATISTICS PROFILE ON	SET STATISTICS XML ON	Include Actual Execution Plan in Management Studio

הבה נתחיל עם הסגנון הפשוט ביותר של showplan.

SET SHOWPLAN_TEXT ו-SHOWPLAN_ALL

להלן דוגמה ל– SHOWPLAN_TEXT עבור join בין שתי טבלאות ממסד הנתונים Northwind:

```
SET NOCOUNT ON;
USE Northwind;
GO
SET SHOWPLAN_TEXT ON;
GO
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 100;
GO
SET SHOWPLAN_TEXT OFF;
GO
```

להלן התוצאה של הרצת קוד זה ב– SQL Server Management Studio :

```
StmtText
------------------------------------------------------------------------
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 100;

StmtText
------------------------------------------------------------------------
  |--Nested Loops(Inner Join, OUTER REFERENCES: ([Northwind].[dbo].
[Products].[ProductID]))
      |--Clustered Index Scan(OBJECT: ([Northwind].[dbo].[Products].
[PK_Products]), WHERE:([Northwind].[dbo].[Products].[UnitPrice]>
($100.0000)))
       |--Index Seek(OBJECT:([Northwind].[dbo].[OrderDetails].
[ProductID]), SEEK:([Northwind].[dbo].[Order Details].[ProductID]=
[Northwind].[dbo].[Products].[ProductID]) ORDERED FORWARD)
```

הפלט אומר לנו שתוכנית השאילתה מורכבת משלושה אופרטורים: Nested Loops, Clustered Index Scan ו– Index Seek. ה– Nested Loops מבצע inner join בין שתי הטבלאות. הטבלה החיצונית (הטבלה הראשונה אליה ניגשים כאשר מבצעים את ה–inner join, נמצאת על הענף העליון הנכנס לאופרטור ה–join) ב–join, היא טבלת Products, ו– SQL Server משתמש ב– Clustered Index Scan כדי לגשת לנתונים הפיסיים. מכיוון שאינדקס–clustered מכיל את כל נתוני הטבלה, סריקת האינדקס–clustered מקבילה לסריקה של הטבלה כולה. הטבלה הפנימית היא טבלת Order Details, אשר לה אינדקס–nonclustered על הטור ProductID, ו– SQL Server משתמש ב– Index Seek כדי לגשת לשורות האינדקס. הפנייה לאובייקט אחרי האופרטור של Index Seek מראה לנו את השם המלא של האינדקס בו השתמשנו: [Northwind].[dbo].[Order Details].[ProductID]. מקרה זה מבלבל מעט מכיוון ששם האינדקס זהה לשם טור ה–join. מסיבה זו, אני ממליץ לא לתת לאינדקסים שם זהה לטורים אליהם הם ייגשו. אופרטור הסריקה עוקב אחר הפנייה לאובייקט. ערך הטור של הטבלה החיצונית משמש לביצוע הסריקה לתוך האינדקס של ה–ProductID.

כאשר התוכנית מופעלת, הזרימה הכללית של השורות היא מלמעלה למטה ומימין לשמאל. האופרטור המוטה יותר מייצר שורות עבור האופרטור המוטה פחות, אשר מפיק שורות עבור האופרטור הבא שמעליו וכך הלאה. במקרה של join, ישנם שני אופרטורי קלט באותה רמה לימין אופרטור ה–join המציינים את שני סטי השורות המאוחדים. לגבוה מבין השניים (Clustered Index Scan במקרה שלנו) מתייחסים כטבלה החיצונית, והנמוך (Index Seek במקרה שלנו) הוא הטבלה הפנימית. הפעולה על הטבלה החיצונית מאותחלת

ראשונה; זו על הטבלה הפנימית מופעלת שוב ושוב לכל שורה מהטבלה החיצונית המגיעה לאופרטור ה–join. בנוסף לאופרטורי ה–join הבינאריים, קיימים גם אופרטורים n-ary בעלי n ענפי קלט — למשל, שרשור בתוכניות עבור שאילתות עם UNION ALL. עבור אופרטורים אלו, הענף העליון מופעל ראשון, אחריו הענף הנמוך יותר וכך הלאה.

שים לב שבדוגמה הקודמת השתמשתי לאחר השאילתה ב– SET SHOWPLAN_TEXT OFF. זאת מכיוון ש– SET SHOWPLAN_TEXT ON לא רק גורם לתוכנית השאילתה להופיע, הוא גם מכבה את הפעלת השאילתה עבור ה–connection. הפעלת השאילתה תישאר כבויה עד ש– SQL Server יפעיל SET SHOWPLAN_TEXT OFF באותו connection. יש לנהוג בזהירות כאשר מבצעים showplan של batch אשר יוצר או משנה אינדקסים או טבלאות קבועות וכתוצאה מכך משתמש בהם בשאילתות באותו batch. מכיוון ש– SQL Server אינו מפעיל את ה–batch אם SHOWPLAN_TEXT דלוק, האובייקטים החדשים ב–batch או אלו שהשתנו אינם מזוהים כאשר פונים אליהם לאחר מכן. כתוצאה מכך הפעולה תכשל אם טבלה קבועה נוצרת ובאה לידי שימוש באותו batch, או שאתה עלול לשגות ולחשוב שהאינדקס החדש שנוצר לא יבוא לידי שימוש בתוכנית השאילתה. יוצאי הדופן היחידים לכלל זה הם טבלאות זמניות ומשתני טבלה שנוצרים לצורך הפקת ה–showplan, אך יצירתם בסופו של דבר עוברת rollback בסיום הפעלת ה–showplan.

SHOWPLAN_ALL דומה מאוד ל–SHOWPLAN_TEXT. ההבדל היחידי הוא המידע הנוסף על תוכנית השאילתה המופק על ידי SHOWPLAN_ALL. הוא מוסיף הערכות של מספר השורות המיוצרות על ידי כל אופרטור בתוכנית השאילתה, הגודל המשוער של שורות התוצאה, זמן המעבד המשוער והעלות הכוללת המשוערת ששימשה פנימית להשוואת תוכנית זו לתוכניות אפשריות אחרות. לא אציג לך את הפלט מ– SHOWPLAN_ALL מכיוון שהוא רחב מדי מכדי להתאים לדף בספר זה. אך המידע המוחזר הוא עדיין מידע חלקי כאשר משווים אותו לפורמט XML של showplan, אותו אתאר כעת.

צורת XML של ה–Showplan

קיימים שני סוגים שונים של XML showplans. אחד, המתקבל על ידי SET SHOWPLAN_XML ON, מכיל תוכנית עבודה משוערת; השני, הפלט של SET STATISTICS XML ON, מכיל גם מידע זמן ריצה. מכיוון שהפלט של SET SHOWPLAN_XML מיוצר על ידי קומפילציה של batch, הוא יפיק מסמך XML יחיד עבור כל ה–batch. מצד שני, הפלט של SET STATISTICS XML מיוצר בזמן ריצה ויתקבל מסמך XML נפרד לכל משפט ב–batch. בנוסף לפקודות הסט, קיימות שתי דרכים נוספות להשגת ה– XML showplan – על ידי שמירת ה–showplan הגרפי המוצג על ידי SSMS, ועל ידי שימוש ב– SQL Server Profiler. את שניהם אתאר בהמשך הפרק.

סכמת XML יחידה, showplanxml.xsd, מכסה הן את ה–showplan המשוער והן את ה– XML showplan של זמן ריצה; עם זאת, פלט זמן הריצה מספק מידע נוסף. לפיכך, האלמנטים והמאפיינים הספציפיים שלו אופציונליים ב–xsd. התקנת SQL Server 2005

ממקמת את הסכמה בתיקייה Microsoft SQL Server\90\Tools\Binn\schemas\sqlserver\ 2004\07\showplan. הסכמה זמינה גם בכתובת http://schemas.microsoft.com/sqlserver.

Showplan במבנה XML יכול להישמר כקובץ ולקבל את הסיומת sqlplan (למשל, batch1.sqlplan). פתיחת קובץ עם סיומת sqlplan תשתמש אוטומטית ב– SQL Server Management Studio (אם מותקן) להצגת showplan גרפי. אין צורך לקשר את השרת בו הופק ה–showplan או את זה שמחזיק את האובייקטים אליהם פונים. מאפיין נהדר זה של SQL Server 2005 מאפשר עבודה עם ה–showplans השמורים והמשותפים (למשל, דרך דואר אלקטרוני), בצורה גרפית ללא אחסנה קבועה של קבצי תמונה גדולים.

XML הוא מבנה ה–showplan העשיר ביותר. הוא מכיל מידע ייחודי שאינו זמין בשום showplan טקסטואלי או גרפי. למשל, רק XML showplan מכיל את גודל התוכנית (המאפיין CachedPlanSize) וערכי פרמטר עבורם התוכנית עברה אופטימיזציה (האלמנט ParameterList), ורק ה– XML showplan של זמן ריצה מכיל את מספר השורות המעובדות ב–threads שונים של תוכנית מקבילית (המאפיין ActualRows של האלמנט RunTimeCountersPerThread) או את מידת המקביליות האמיתית כאשר השאילתה הוצאה לפועל (המאפיין DegreeOfParallelism של התוכנית – DOP).

כפי שהסברתי קודם לכן, XML showplan יחיד יכול להכיל מידע לגבי מספר משפטים ב–batch. עליך לזכור זאת כאשר אתה מפתח תוכנה המעבדת XML showplan. עליך לחשוב על מקרים של batches מרובי משפטים ולכלול אותם בבדיקות שלך אלא אם כן תעבד אך ורק מסמכי XML showplan המיוצרים על ידי ה– SET STATISTICS XML או על ידי SQL Server Profiler.

אולי היתרון הגדול ביותר של מבנה ה–XML הוא שהוא יכול להיות מעובד על ידי שימוש בכל טכנולוגיית XML – למשל, XPath, XQuery או XSLT.

מידע נוסף: דוגמה טובה לחילוץ נתונים מה- XML showplan ניתן למצוא בכתובת: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/xmlshowplans.asp.

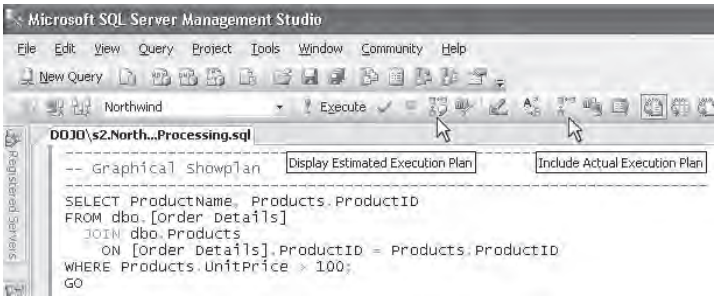
מאמר זה מתאר יישום המחלץ את עלות ההפעלה המשוערת של שאילתה מה- XML showplan שלה. על ידי שימוש בשיטה זו, משתמש יכול להגביל את שליחת השאילתות רק לאלו שעולות פחות מרמת סף הנקבעת מראש. דבר זה יבטיח ששאילתות בעלות זמן ריצה ארוך לא יעמיסו על השרת.

אני משוכנע כי ה– XML showplan יוביל לפיתוח של כלים רבים שיסייעו למנהלי מערכת, מפתחים ועובדי תפעול בעבודתם היומיומית. שאילתות חדשות אשר עוזרות לנתח את ה– XML showplans מופיעות באינטרנט בתדירות גדלה והולכת.

גרפי Showplan

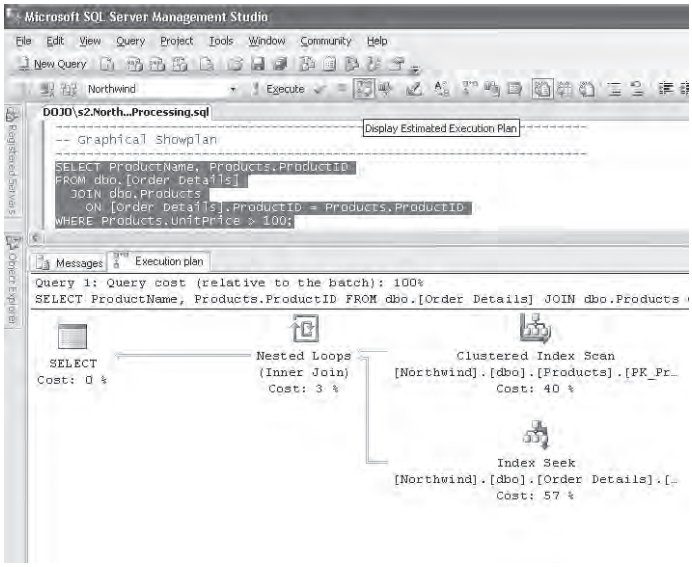
ל–SSMS קיימות שתי אפשרויות להצגת המבנה הגרפי של showplan: האפשרות Display Estimated Execution Plan והאפשרות Include Actual Execution Plan, אותן ניתן למצוא בתפריט Query ובלחצנים בסרגל הכלים כפי שניתן לראות בתרשים 8–2.

תרשים 8–2: האפשרויות Display Estimated Execution Plan ו– Include Actual Execution Plan ב-SSMS



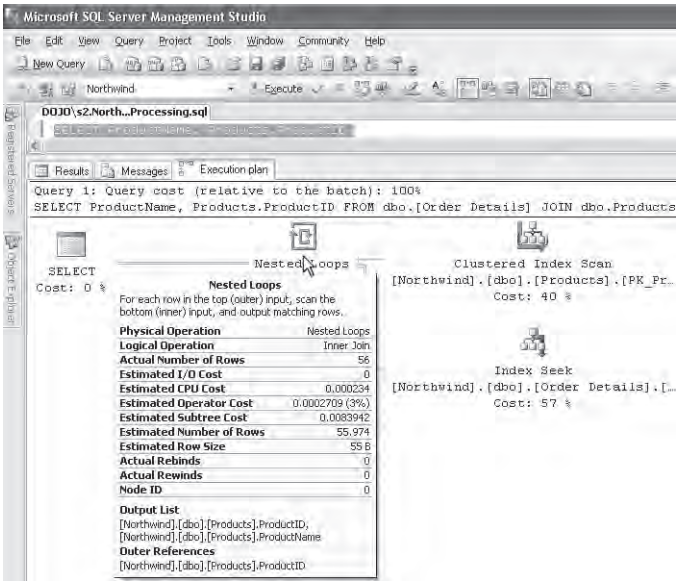
קיים הבדל משמעותי בין האפשרות Display Estimated Execution Plan לבין האפשרות Include Actual Execution Plan. אם תבחר בראשונה, תמונה המציגה את ה–showplan הגרפי של השאילתה או של ה–batch בחלון השאילתה תופיע כמעט מיידית (במהירות התלויה בזמן הקומפילציה ובשאלה האם התוכנית נמצאת כבר ב–cache או לא) תחת הכרטיסייה Execution Plan בחלק התוצאה של SSMS, כפי שמוצג בתרשים 9–2.

תרשים 9–2: Display Estimated Execution Plan

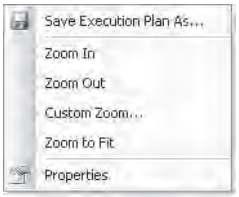


הפעלת הלחצן Include Actual Execution Plan אינה מבצעת שום פעולה מיידית. היא רק משנה את המצב של SSMS לכלול את ה–showplan של זמן ריצה בכל הפעלת משפט או batch. הפעלתי את הלחצן, ואז הרצתי את אותה שאילתה כמקודם על ידי לחיצה על הלחצן Execute בעל סימן הקריאה האדום. כרטיסיית Result הנוספת הופיעה בין חלונות התוצאה לצד הכרטיסיות Messages ו– Execution Plan בחציו התחתון של חלון ה–SSMS. אם תבחר בכרטיסיות אחת אחר השנייה, תראה שחלון ה–Results מכיל את תוצאת השאילתה ובחלון Execution Plan יש תוכנית הדומה לזו שזה עתה הוצגה בתרשים 9–2. ההבדלים בתוכניות נראים לעין רק לאחר שאתה חוקר את תוכן המידע בתוך האופרטורים השונים. בתרשים 10–2 ריחפתי עם סמן העכבר מעל האופרטור Nested Loops, אשר העלה מידע נוסף שאינו קיים בתוכנית המשוערת – שדות וערכים עבור "Actual Number Of Rows", "Actual Rebinds" ו–"Actual Rewinds" (כולם יוסברו בהמשך).

תרשים 10–2: Include Actual Execution Plan



תרשים 11–2: אפשרויות תוכנית הפעלה



השורות זורמות מימין לשמאל, וכאשר מצרפים שתי טבלאות, הטבלה החיצונית נמצאת מעל הטבלה הפנימית ב–showplan הגרפי.

אם תלחץ עם המקש הימני של העכבר בחלון Execution Plan, יופיע חלון צץ עם האפשרויות המוצגות בתרשים 2–11. השתמשתי באפשרות Zoom To Fit להפקת תמונות התוכנית בתרשימים 2–9 ו–2–10. האפשרות Properties תציג מאפיינים של האופרטור בו בחרת לפני שלחצת על המקש הימני של העכבר, בדומה לדוגמה שהוצגה עבור האופרטור Nested Loops בתרשים 2–10. הפעולה שהיא אולי הכי פחות מובנית מאליה קשורה לאפשרות Save Execution Plan As. אם תבחר באפשרות זו, SSMS יבקש ממך מיקום קובץ בו יאוחסן ה– XML showplan (לא הייצוג הגרפי של התוכנית). סיומת ברירת המחדל עבור שם הקובץ היא sqlplan.

מידע זמן ריצה ב–Showplan

SQL Server אוסף את מידע זמן הריצה במהלך הרצת השאילתה. כפי שהסברתי קודם לכן, XML showplan הוא המבנה השלם ביותר של תוכנית שאילתה. לפיכך, נתחיל ראשית עם בחינת מידע זמן הריצה הנוסף המוחזר ב– XML showplan. לאחר מכן נדון בפלט של הפקודה SET STATISTICS PROFILE, ולבסוף, נדון במידע ה–showplan בזמן הריצה של SQL Server Profiler.

SET STATISTICS XML ON|OFF

קיימים שני סוגים של מידע זמן ריצה ב– XML showplan: עבור משפט SQL ועבור thread. אם למשפט יש פרמטר, התוכנית מכילה את המאפיין ParameterRuntimeValue, המציג את הערך לכל פרמטר כאשר המשפט מופעל. ערך זה עשוי להיות שונה מהערך המשמש לביצוע קומפילציה של המשפט תחת המאפיין ParameterCompiledValue, אך מאפיין זה קיים בתוכנית רק אם ה–optimizer יודע את הערך של הפרמטר בזמן ביצוע האופטימיזציה והוא true רק עבור פרמטרים המועברים לפרוצדורות מאוחסנות.

בשלב הבא, יש את המאפיין DegreeOfParallelism, המציג את רמת המקביליות בפועל (או DOP, שהוא מספר ה–threads הפועלים בו זמנית על השאילתה הבודדת) של ההרצה. זה, שוב, עשוי להיות שונה מהערך בזמן קומפילציה. הערך בזמן קומפילציה אינו מופיע בתוכנית השאילתה, אך הוא תמיד שווה למחצית מספר המעבדים הזמינים ל– SQL Server אלא אם כן מספר המעבדים הוא 2 – במקרה זה, ערך DOP בזמן קומפילציה יהיה 2 גם הוא. ה–optimizer לוקח בחשבון מחצית מהמעבדים מכיוון שה–DOP בזמן קומפילציה יעבור כוונון בהתבסס על עומס העבודה בזמן שההרצה מתחילה; הוא עשוי לקבל כל מספר בין 1 לבין מספר המעבדים. ללא קשר לבחירה הסופית עבור DOP, נעשה שימוש באותה תוכנית מקבילית. אם תוכנית מקבילית מופעלת בסופו של דבר עם DOP = 1, SQL Server יסיר את אופרטורי ההחלפה מתוכנית השאילתה כאשר הוא ייצור את

הקשר ההרצה. המאפיין MemoryGrant מציג זיכרון בפועל הניתן לשאילתה עבור הרצה באלפי בתים (Kilobytes). SQL Server משתמש בזיכרון זה לבניית טבלאות hash עבור hash joins או לביצוע מיון בזיכרון.

האלמנט RunTimeCountersPerThread מכיל חמישה מאפיינים, כל אחד מהם עם ערך אחד לכל thread: ActualRebinds, ActualRewinds, ActualRows, ActualEndofScans ו–ActualExecution. פרק 3 מתאר כיצד SSMS מציג Actual Number Of Rows, Actual Rebinds ו–Actual Rewinds בחלון המידע הצץ של האופרטור ב–showplan הגרפי. החלון הצץ מציג ערכים מצטברים (מתווספים לאורך כל ההרצות של כל ה–threads) לכל אחד מהערכים ActualRows, ActualRebinds ו– ActualRewinds מה– XML showplan. הערך ActualExecutions אומר לנו כמה פעמים האופרטור אותחל בכל אחד מה–threads. אם האופרטור הוא אופרטור סריקה, הספירה ActualEndofScans מראה כמה פעמים הסריקה מגיעה לסוף הסט. כתוצאה מכך, הפחתת ActualEndofScans מ– ActualExecutions אומרת לנו כמה פעמים האופרטור לא סרק את כל הסט – דבר זה עשוי לקרות, למשל, אם TOP ב–SELECT מגביל את מספר השורות המוחזרות וסט הפלט נאסף בטרם הסריקה מגיעה לקצה הטבלה. בדומה, במקרה של Merge Join, אם אחד הסטים מוצה אין צורך להמשיך לסרוק את הסט הבא מכיוון שלא יכולות להיות עוד התאמות.

ה– XML showplan עשוי להכיל גם הזהרות. אלו אירועים שנוצרו במהלך קומפילציה או במהלך הרצה. דוגמאות לאזהרות הנוצרות על ידי הקומפיילר הן סטטיסטיקות חסרות וביטוי join חסר. דוגמאות לאזהרות זמן–ריצה הן hash bailout ו– exchange spill. אם נתקלת באזהרה בתוכנית השאילתה שלך, עליך לפנות למידע נוסף ל– "Errors and Warnings Event Category" ב– Books Online.

SET STATISTICS PROFILE

SET STATISTICS PROFILE ON מחזיר מידע דומה ל– SET SHOWPLAN_ALL ON. עם זאת, קיימים שני הבדלים: SET STATISTICS PROFILE פעיל בעת הרצת משפטים, זה הזמן בו הוא מפיק מידע נוסף לצד תוצאת השאילתה עצמה (המשפט חייב לסיים את ההרצה בטרם הפלט הנוסף מופק). כמו כן, הוא מוסיף שני טורים לפלט: Rows ו–Executes. ערכים אלו נובעים מהפלט של ה– XML showplan של זמן ריצה. הערך Rows הוא הסכום של המאפיין RowCount באלמנט RunTimeCountersPerThread על פני כל ה-threads. (קיימים threads מרובים רק אם זו תוכנית מקבילית ובפועל רצה בצורה מקבילית). הערך Execute הוא הסכום של המאפיין ActualExecution באותו אלמנט. טבלה 4–2 מציגה דוגמה לחלק מקוצר מהפלט של SET STATISTICS PROFILE עבור אותה שאילתה עליה עבדנו קודם.

טבלה 4–2: פלט של STATISTICS PROFILE

Rows	*Executes*	*StmtText*
56	1	SELECT ProductName, Products.ProductId
56	1	|--Nested Loops(Inner Join, OUTER REFERENCES:([Northwind]
2	1	|--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Products].[
56	2	|--Index Seek(OBJECT:([Northwind].[dbo].[OrderDetails].[Produ

הטור Rows מכיל את מספר השורות המוחזרות בפועל על ידי כל אופרטור. המספר עבור Executes אומר לנו כמה פעמים SQL Server איתחל את האופרטור לבצע עבודה על שורה אחת או יותר. מכיוון שהצד החיצוני של ה–join (ה–Clustered Index Scan של טבלת Products) החזיר שתי שורות, היה עלינו להריץ את הצד הפנימי של ה– join (Index Seek) פעמיים. לפיכך, Index Seek מקבל את המספר 2 בטור Executes בפלט.

כאשר בוחנים את התוכנית עבור שאילתה מסוימת, דרך טובה למציאת בעיות פוטנציאליות היא למצוא את אי ההתאמות הגדולות ביותר בין ההערכות של ה–optimizer לבין המספר האמיתי של הרצות ושורות מוחזרות. כאן עלינו להיות זהירים מכיוון שההערכה של ה–optimizer בטור EstimateRows היא עבור כל הרצה משוערת, בעוד שה–Rows בפלט של ה–showplan שהוזכר קודם הוא מספר השורות המצטבר המוחזר על ידי האופרטור מכל ההרצות שלו. לפיכך, כדי להעריך את אי ההתאמות של ה–optimizer עלינו להכפיל את EstimateRows ב– EstimateExecution ולהשוות את התוצאה עם המספר בפועל של כל השורות המוחזרות בטור Rows בפלט של SET STATISTICS PROFILE.

במקרים מסוימים, ניתן לייצר תוכניות על ידי שימוש ב– SET STATISTICS PROFILE ו– SET STATISTICS XML עבור משפטים עבורם ה– SET SHOWPLAN אינו מפיק כל פלט. דוגמאות לכך הן CREATE INDEX על טבלה לא ריקה, sp_executesql ו–batch היוצר ופונה לאותו אובייקט.

לכידת Showplan עם SQL Trace

SQL Server Profiler הוא כלי GUI המגדיר ולוכד אירועי trace של SQL Server 2005 שהתקבלו משרת. SQL Server מציג את האירועים בחלון ה–Profiler ויכול אופציונלית לשמור אותם בקובץ trace או בטבלה שיכולה מאוחר יותר לעבור ניתוח או לשמש להרצה חוזרת של סדרת צעדים מסוימת כאשר מנסים לאבחן בעיה. בפרק 3 יוסבר מדוע שימוש בקוד T-SQL להגדרת ה– trace יעיל יותר מאשר שימוש ב– Profiler GUI, ומדוע יש להשתמש ב–T-SQL כאשר מנטרים עומסי עבודה לא רגילים. המידע להלן בנוגע ללכידת ה–showplan רלוונטי לשימוש הן ב–GUI והן בקבוצת פרוצדורות ה–T-SQL המאוחסנות המתחילות ב– sp_trace_.

שימוש ב–trace ללכידת מידע ה–showplan הוא מדויק ביותר מכיוון שאתה נמנע מאי התאמות נדירות אך אפשריות בין ה–showplan אותו אתה בוחן ב–SSMS והתוכנית בפועל שהייתה בשימוש במהלך ההרצה של היישום שלך. אפילו אם אינך משנה את ה–metadata (הוספה או הסרה של אינדקסים, עדכון אילוצים, יצירת או עדכון סטטיסטיקות), אתה עדיין עשוי להיתקל במקרים בהם יש לך תוכנית שונה בזמן ההרצה מאשר בזמן הקומפילציה המקורית. המקרים הנפוצים ביותר קורים כאשר פרוצדורה מאוחסנת אחת נקראת עם ערכי פרמטר אחרים, כאשר סטטיסטיקות מעודכנות אוטומטית, וכאשר ישנו שינוי במשאבים זמינים (מעבד וזיכרון) בין זמן הקומפילציה לבין זמן ההרצה. ניטור עם trace דורש משאבים רבים, והוא עלול להשפיע בצורה שלילית על הביצועים של השרת שלך. ככל שאתה מנטר יותר אירועים, כך גדלה ההשפעה. לפיכך, עליך לבחור את האירועים אותם אתה מנטר בזהירות ולחלופין עליך לשקול חילוץ showplans מ–cache הפרוצדורות, כפי שאתאר בהמשך. החיסרון בשימוש ב–cache הפרוצדורות הוא חוסר היציבות שלו (אם השאילתה אינה מורצת, תוכנית השאילתה עשויה להיות מוסרת מה–cache) וחוסר מידע זמן ריצה בנוגע להרצות שאילתה בודדות.

קיימות תשע מחלקות אירועים הלוכדות סוגים שונים של מידע showplan תחת קטגוריית האירועים: Performance. טבלה 5–2 תסייע לך לבחור את האירוע המתאים ביותר לצרכיך.

טבלה 5–2: מחלקות אירועי trace הקשורות ל–Showplan

מחלקת אירוע trace	קומפילציה או ריצה	כולל מידע זמן ריצה	XML כולל showplan	מול trace מפיק SQL Server 2000
Showplan All	ריצה	לא	לא	כן
Showplan All for Query Compile	קומפילציה	לא	לא	לא
Showplan Statistics Profile	ריצה	כן[1]	לא	כן
Showplan Text	ריצה	לא	לא	כן
Showplan Text (Unencoded)	ריצה	לא	לא	כן[2]
Showplan XML	ריצה	לא	כן	לא
Showplan XML for Query Compile	קומפילציה	לא	כן	לא

מחלקת אירוע trace	קומפילציה או ריצה	כולל מידע זמן ריצה	XML כולל showplan	מול trace מפיק SQL Server 2000
Showplan XML Statistics Profile	ריצה	כן[3]	כן	לא
Performance Statistics	קומפילציה וריצה [4]	כן[5]	כן	לא

1 מידע זמן הריצה המופק זהה לזה המופק על ידי SET STATISTICS PROFILE ON.

2 אם ה–Profiler של SQL Server 2005 מקושר לשרת SQL Server 2000, הוא אוטומטית מראה רק את אירועי ה–showplan הנתמכים על ידי SQL Server 2000. שים לב שהאירוע "Showplan Text (Unencoded)" נקרא "Execution plan" ב– SQL Server 2000.

3 הטור TextData בפלט ה–profiler מכיל את ה– XML showplan, כולל אותו מידע זמן ריצה כמו בפלט של SET STATISTICS XML ON.

4 מחלקת האירועים Performance Statistics היא שילוב של מספר תת–אירועים. תת–אירוע 1 ו–2 מפיקים את אותו showplan כמו האירוע Showplan XML For Query Compile – 1 עבור פרוצדורות מאוחסנות, ו–2 עבור משפטי אד–הוק. תת–אירוע 3 מפיק סטטיסטיקות זמן ריצה מצטברות עבור השאילתה כאשר היא מוסרת מ–cache הפרוצדורות.

5 זהו מידע זמן ריצה מצטבר עבור כל ההפעלות של שאילתה זו שהופק עבור תת–אירוע 3. המידע שנלכד זהה לזה המופק על ידי ה– sys.dm_exec_query_stats DMV, והוא מאוחסן בפורמט XML בטור TextData בפלט של ה–trace.

לשם הפשטות, אגביל את המשך הדיון ל–trace ב– SQL Server 2005 בלבד.

אם השרת שלך אינו עמוס מאוד או שהוא משמש כמערכת פיתוח או בדיקות, עליך להשתמש באירוע Showplan XML Statistics Profile. הוא מייצר את כל מידע תוכנית השאילתה וזמן ריצה שאתה עשוי להצטרך. אתה יכול גם לעבד את תוכניות העבודה בצורה תכנותית או להציג אותן בצורה גרפית.

אפילו אם השרת שלך עמוס אך קצב הקומפילציה נשמר ברמה נמוכה על ידי שימוש חוזר יעיל בתוכניות, תוכל להשתמש באירוע Showplan XML For Query Compile מכיוון שהוא מפיק רשומות trace רק כאשר פרוצדורה מאוחסנת או משפט עוברים קומפילציה או קומפילציה חוזרת. trace זה לא יכיל מידע זמן ריצה.

האירוע Showplan Text מייצר פלט זהה לזה של האירוע Showplan Text (Unencoded) אלא שה–showplan מאוחסן כמחרוזת תווים בטור TextData עבור האירוע השני וכנתון בינארי מקודד בטור BinaryData עבור הראשון. תוצאה של שימוש בטור Binary Data היא שהוא אינו ניתן לקריאה על ידי בני אדם – אנו זקוקים ל–Profiler או לפחות ל–DLL המקודד של Profiler כדי לקרוא את ה–showplan. המבנה הבינארי מגביל גם את מספר הרמות במסמך ה–XML ל–128. מצד שני, הוא דורש פחות מקום לאחזקת כל מידע התוכנית.

תוכל להקטין את גודל ה– trace על ידי בחירה בערכי סינון לטורים שונים. גישה טובה היא לייצר trace ניסיוני קטן, לנתח אילו אירועים וטורים אינך צריך, ולהסיר אותם מה– trace. על ידי שימוש בתהליך זה, באפשרותך גם לזהות מידע חסר שתרצה להוסיף ל– trace. כאשר אתה בונה את מסנן ה– trace (למשל, על ידי שימוש ב– sp_trace_setfilter), רק מסננים על הטורים ApplicationName, ClientProcessID, HostName, LoginName, LoginSid, NTDomainName, NTUserName ו–SPID מנטרלים את יצירת האירוע. יתר המסננים מופעלים רק לאחר שהאירוע נוצר ונשלח ללקוח. לפיכך, לא רצוי להחליף סינון נרחב בבחירה זהירה של אירועים; הסינון עשוי פוטנציאלית לגרום להסרת כל הרשומות של מחלקת אירוע מסוימת מה– trace מבלי שיסייע להקטין את ההשפעה על השרת. למעשה, יהיה צורך ביותר עבודה ולא בפחות.

בהשוואה לאפשרויות SET STATISTICS PROFILE ו– SET STATISTICS XML, אירועי ה– trace של showplan מגדילים עוד יותר את סט המשפטים עבורם SQL Server לוכד את התוכנית. מחלקות המשפטים הנוספות הן auto, non-auto-CREATE ו– UPDATE STATISTICS, כמו גם המשפט EXEC … INSERT INTO.

חילוץ ה–Showplan מ–Cache הפרוצדורות

לאחר שה– query optimizer מייצר את התוכנית עבור ה–batch או ה– stored procedure, התוכנית ממוקמת ב–cache הפרוצדורות. תוכל לבחון את cache הפרוצדורות על ידי שימוש במספר DMV (Dynamic Management Views) ו– DMF (Dynamic Management Functions), DBCC PROCCACHE, וה– sys.syscacheobjects catalog view (שנשאר לצורך תאימות אחורנית). כעת אציג כיצד ניתן לגשת ל–showplan בפורמט XML עבור השאילתות הנמצאות כרגע ב–cache הפרוצדורות.

ה– DMF sys.dm_exec_query_plan מחזיר את ה–showplan בפורמט XML לכל שאילתה אשר תוכנית העבודה שלה נמצאת כרגע ב–cache הפרוצדורות. ה– DMF sys.dm_exec_query_plan דורש מזהה תוכנית (plan handle) כארגומנט היחיד שלו. מזהה התוכנית הוא מזהה מסוג VARBINARY (64) של תוכנית השאילתה, וה– DMV sys.dm_exec_query_stats מחזיר אותו עבור כל שאילתה הנמצאת כרגע ב–cache הפרוצדורות. השאילתה הבאה מחזירה את ה– XML showplans עבור כל תוכניות השאילתה ב–cache. אם batch או פרוצדורה מאוחסנת מכילים מספר משפטי SQL

עם תוכנית שאילתה, ה–view יכיל שורה נפרדת לכל אחד מהם.

```
SELECT qplan.query_plan AS [Query Plan]
FROM sys.dm_exec_query_stats AS qstats
 CROSS APPLY sys.dm_exec_query_plan(qstats.plan_handle) AS qplan;
```

קשה למצוא את תוכנית השאילתה עבור שאילתה מסוימת על ידי שימוש בשאילתה לעיל מכיוון שמלל השאילתה מוכל רק עמוק בתוך ה– XML showplan. ההרחבה הבאה לשאילתה הקודמת מחלצת את המספר הסידורי (הטור שנקרא No) ואת מלל השאילתה (הטור Statement Text) מה–showplan על ידי שימוש בשיטת ערך ה–Xquery. לכל batch יש sql_handle יחיד; לפיכך, הגדרת ORDER BY sql_handle, [No] מבטיחה ששורות פלט עבור ה–batch המכיל מספר משפטי SQL מוצגות אחת אחר השנייה בסדר בו הן מופיעות ב–batch.

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/
showplan' AS sql)
SELECT
  C.value('@StatementId','INT') AS [No],
  C.value('(./@StatementText)','NVARCHAR(MAX)') AS [Statement Text],
  qplan.query_plan AS [Query Plan]
FROM (SELECT DISTINCT plan_handle FROM sys.dm_exec_query_stats) AS qstats
  CROSS APPLY sys.dm_exec_query_plan(qstats.plan_handle) AS qplan
  CROSS APPLY query_plan.nodes('/sql:ShowPlanXML/sql:BatchSequence/
sql:Batch/sql:Statements/descendant::*[attribute::StatementText]')
    AS T(C)
ORDER BY plan_handle, [No];
```

כעת, אציג חלק מהתוצאה המוחזרת על ידי השאילתה לעיל. הפלט תלוי בתוכן הנוכחי של cache הפרוצדורות; לפיכך, הפלט שלך יהיה שונה כמעט בוודאות מזה המוצג בטבלה 6–2.

טבלה 6–2: מידע עבור תוכניות שאילתה המחולצות מה–Cache

No	Statement Text	Query Plan
1	SELECT CAST(serverproperty(N'S	<ShowPlanXML xmlns="http://sch
1	select value_in_use from sys.c	<ShowPlanXML xmlns="http://sch
1	with XMLNAMESPACES ('http://sc	<ShowPlanXML xmlns="http://sch
1	with XMLNAMESPACES ('http://sc	<ShowPlanXML xmlns="http://sch
1	IF (@@microsoftversion / 0x010	<ShowPlanXML xmlns="http://sch
2	SELECT se.is_admin_endpoint A	<ShowPlanXML xmlns="http://sch
3	ELSE	<ShowPlanXML xmlns="http://sch
1	SELECT CAST(serverproperty(N'S	<ShowPlanXML xmlns="http://sch

תוכניות עדכון

ה– query optimizer חייב לטפל במספר סוגיות ספציפיות כאשר הוא מבצע אופטימיזציה למשפטי INSERT, UPDATE ו–DELETE – או, במילים אחרות, **שינוי נתונים**. כאן אתאר את השיטות המשמשות את SQL Server לעיבוד משפטים אלו.

לתוכניות ה–IUD (קיצור בו אשתמש עבור "INSERT, UPDATE ו–DELETE") שני שלבים. השלב הראשון הוא **read only**, והוא קובע אילו שורות צריכות להתווסף/להתעדכן/להימחק על ידי יצירת זרם נתונים המתאר את השינויים שיש לבצע. עבור INSERTs, זרם הנתונים מכיל ערכי טור; עבור DELETEs יש לו את מפתחות הטבלה; ועבור UPDATEs, יש לו הן את מפתחות הטבלה והן את ערכי הטורים שהשתנו. השלב השני מיישם על הטבלה שינויים בזרם הנתונים; בנוסף, הוא מבצע פעולות הכרחיות לשמירה על אמינות הנתונים על ידי ביצוע בדיקת אילוצים, הוא מתחזק את אינדקסים–nonclustered ו– indexed views, והוא מפעיל טריגרים אם קיימים. בדרך כלל תוכניות השאילתה ל–UPDATE ו–DELETE מכילות שתי פניות לטבלת היעד: הפנייה הראשונה משמשת לזיהוי השורות המושפעות, והשנייה משמשת לביצוע השינוי. תוכניות ה–INSERT מכילות רק פנייה אחת לטבלת היעד, אלא אם כן אותה טבלת יעד משתתפת גם ביצירת השורות המוכנסות.

במקרים פשוטים, SQL Server ממזג את שלבי הקריאה והכתיבה של תוכניות ה–IUD. זה המקרה, למשל, כאשר מכניסים ערכים ישירות לתוך טבלה (תהליך הנקרא scalar insert) או כאשר מעדכנים/מוחקים שורות המזוהות על ידי ערך של מפתח עיקרי בטבלת היעד.

האופרטור Assert כלול אוטומטית בתוכניות השאילתה בשלב השני אם SQL Server צריך לבצע בדיקת אילוצים. SQL Server בודק את אילוצי ה–CHECK עבור INSERTs ו–UPDATEs על ידי הערכת ביטוי סקלארי, לא יקר בדרך כלל, לכל שורה וטור מושפעים. אילוצים של מפתח זר (foreign key) מופעלים על INSERTs ו–UPDATEs לטבלה המכילה את אילוץ המפתח הזר, והם מופעלים על UPDATEs ו–DELETEs לטבלה המכילה את המפתח אליו פונים. הטבלה הקשורה שאינה היעד של פעולת ה–IUD נסרקת לווידוא האילוץ; לפיכך, מעורבת גישה לנתונים. הצהרה על מפתח ראשי יוצרת אוטומטית אינדקס ייחודי על טורי המפתח, אך זה לא המקרה עבור מפתח זר. UPDATEs ו–DELETEs של מפתחות ראשיים חייבים לגשת לטבלת המפתחות הזרים לכל עדכון או מחיקה של ערך מפתח ראשי, או לצורך וידוא אי–קיום של המפתח המוסר, או לצורך הפצת השינוי אם המפתח הזר הוגדר עם אפשרות cascade. לפיכך, עליך להבטיח שקיים אינדקס על המפתח הזר אם בכוונתך לבצע UPDATEs המשפיעים על ערכי המפתח או DELETEs מהטבלה הראשית.

בנוסף לביצוע פעולת ה–IUD על האינדקס–clustered או על ה–heap, העיבוד של שאילתות INSERT ו–DELETE דורש גם תחזוקה של כל האינדקסים–nonclustered, ושאילתות ה–UPDATE מתחזקות אינדקסים המכילים את הטורים המעודכנים. מכיוון שאינדקסים–nonclustered מכילים את מפתחות האינדקס–clustered ואת מפתחות ה–partitioning כדי לאפשר גישה יעילה לשורת הטבלה, עדכון טורים המשתתפים

באינדקס–clustered או במפתח ה–partitioning הוא יקר מכיוון שהוא דורש תחזוקה של כל האינדקסים. עדכון מפתח ה–partitioning עלול גם לגרום לשורות לנוע בין מחיצות. לפיכך, כאשר יש לך ברירה, בחר מפתחות partitioning ו–clustering שאין בכוונתך לעדכן.

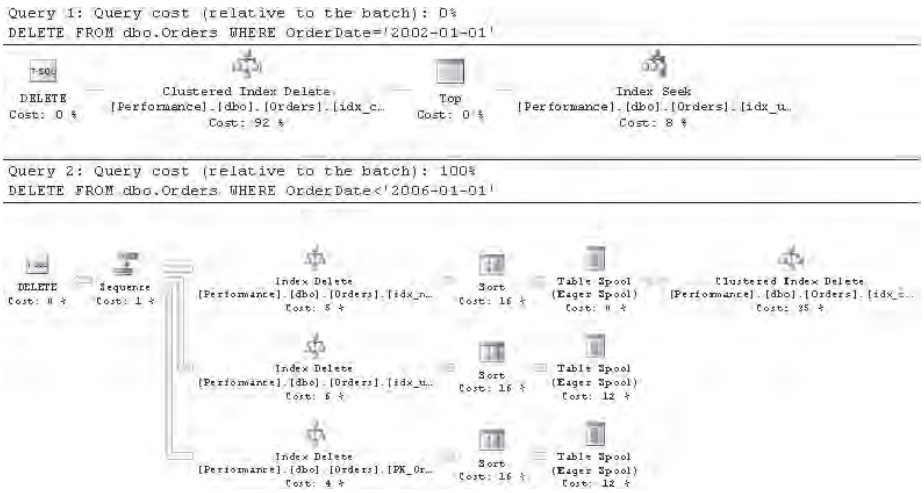
שים לב: SQL Server 2005 מגביל את מפתחות ה–partitioning לטור יחיד; לפיכך, "מפתח ה–partitioning" וטור ה–"partitioning" הן מילים נרדפות.

באופן כללי, הביצועים של משפטי IUD קשורים מאוד למספר האינדקסים המתוחזקים המכילים את טורי היעד, מכיוון שאלו חייבים כולם לעבור עדכון. ביצוע פעולות INSERT ו–DELETE על שורה בודדת לאינדקס דורש מעבר בודד על עץ האינדקס. SQL Server מבצע עדכון לאינדקס או למפתח ה–partitioning כ–DELETE ולאחריו INSERT – לפיכך, המחיר יקר בערך פי שתיים מאשר UPDATE של ערך שאינו מפתח.

ה– query optimizer שוקל ומעריך עלות של שתי אסטרטגיות שונות עבור משפטי IUD: תחזוקה לפי–שורה ולפי–אינדקס. דוגמאות לשתי אסטרטגיות אלו ניתן לראות בתוכניות לשאילתות 1 ו–2 בהתאמה, בתרשים 12–2. עבור תחזוקה לפי–שורה, SQL Server מתחזק את האינדקסים ואת טבלת הבסיס יחד לכל שורה המושפעת מהשאילתה. העדכונים לכל האינדקסים–nonclustered מבוצעים יחד עם כל עדכון שורה בודדת בטבלת הבסיס (שעשויה להיות heap או אינדקס–clustered). התוכנית עבור שאילתה 1 (העליונה בתרשים 12–2) DELETE FROM dbo.Orders WHERE OrderDate='2002-01-01' היא דוגמה לתחזוקה לפי–שורה. שאילתה 1 מוחקת רק 24 שורות בטבלת Orders במסד הנתונים Performance המיוצר על ידי קטע–קוד 1–3 בפרק 3. התוכנית עבור שאילתה 1 אינה מראה שום מחיקות שבוצעו על האינדקסים המשניים מכיוון שהן בוצעו ביחד עם המחיקות של שורות האינדקס–clustered שורה בכל פעם.

התוכנית עבור שאילתה 2 DELETE FROM dbo.Orders WHERE OrderDate<'2006-01-01' בחלק התחתון של תרשים 12–2 תמחק 751,216 שורות מול אותה טבלת Orders, והתוכנית שלה שונה מאוד מכיוון שהיא מבצעת תחזוקת לפי–אינדקס. ראשית, התוכנית מוחקת את השורות הדרושות מהאינדקס–clustered (מצוינות על ידי הסמל Clustered Index Delete מימין) ובו בזמן בונה טבלה זמנית המכילה את ערכי המפתח עבור שלושת האינדקסים –nonclustered שחייבים להיות מתוחזקים. SQL Server קורא את הנתונים הזמניים שלוש פעמים, פעם אחת לכל אחד מהאינדקסים. בין קריאת הנתונים הזמניים לבין מחיקת השורות מהאינדקס–nonclustered, SQL Server ממיין את הנתונים לפי סדר האינדקס המתוחזק, וכך מבטיח גישה אופטימלית לדפי האינדקס.

תרשים 12–2: תוכניות עדכון לפי–שורה ולפי–אינדקס



האופרטור Sequence אוכף את סדר ההרצה של הענפים שלו. SQL Server מעדכן את האינדקסים אחד אחר השני מראש התוכנית ועד לתחתית.

האסטרטגיה לפי–שורה יעילה במונחים של מעבד מכיוון שנדרש מסלול קוד קצר כדי לעדכן את הטבלה ואת כל האינדקסים ביחד. הקוד עבור תחזוקה לפי–אינדקס הוא מעט מורכב יותר, אך עשוי להיות חיסכון משמעותי ב–I/O. על ידי עדכון האינדקסים–nonclustered כל אחד בנפרד לאחר מיון המפתחות, לעולם לא נבקר דף אינדקס יותר מפעם אחת אפילו אם שורות רבות מעודכנות באותו דף. לפיכך, בדרך כלל נבחרת תוכנית עדכון לפי–אינדקס כאשר מעודכנות שורות רבות וה–optimizer מעריך שאותו דף של האינדקס המתוחזק ייקרא יותר מפעם אחת כדי להשלים את התחזוקה על ידי שימוש באסטרטגיה לפי–שורה.

בנוסף לכך שתתקל בנתונים הזמניים המחזיקים את המפתחות לתחזוקת אינדקסים, אתה עשוי גם להיתקל בתוכניות IUD באופרטור הזמני המיוחד המספק "הגנת Halloween" הידועה גם כ– Halloween spool. (את מקור השם אסביר בהמשך הפרק). ה– query optimizer מזריק אופרטור זמני למספר תוכניות IUD כדי להבטיח את נכונות התוצאה המופקת. אשתמש בדוגמה הקטנה הבאה להצגת הבעיה. בטבלת Tiny_employees שלי יש שני טורים – name ו–salary – ובתחילה אינדקס–nonclustered אחד על הטור name. הרץ את הקוד בקטע–קוד 3–2 ליצירת טבלת Tiny_employees ומילויה בנתונים.

קטע–קוד 3–2: קוד למסד הנתונים Halloween

```
SET NOCOUNT ON;
USE master;
GO
IF DB_ID('Halloween') IS NULL
  CREATE DATABASE Halloween;
GO
USE Halloween;
GO

-- Creating and Populating the Tiny_employees Table
IF OBJECT_ID('dbo.Tiny_employees') IS NOT NULL
  DROP TABLE dbo.Tiny_employees;
GO
CREATE TABLE dbo.Tiny_employees (name CHAR(8), salary INT);
INSERT INTO dbo.Tiny_employees VALUES ('emp_A',30000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_B',20000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_C',19000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_D',8000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_E',7500);
GO
CREATE INDEX ind_name ON dbo.Tiny_employees(name);
GO
```

כעת שקול מימוש הבקשה הבאה – הגדל את המשכורת ב–10 אחוזים לכל העובדים בעלי משכורת נמוכה מ–25,000. השאילתה פשוטה:

```
UPDATE dbo.Tiny_employees
  SET salary = salary * 1.1
WHERE salary < 25000;
```

תוכנית השאילתה שלה, המוצגת בתרשים 2–13, משתמשת בתחזוקה לפי–שורה. (אינך רואה צומת עדכון עבור האינדקס ind_name בשום מקום בתוכנית).

תרשים 2–13: תוכנית עבודה עבור המשפט UPDATE

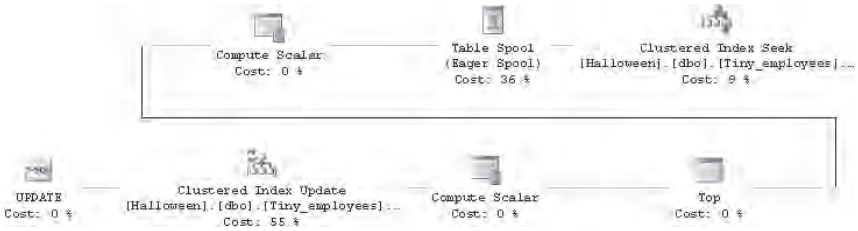
כעת הבא ניצור אינדקס–clustered על הטבלה Tiny_employees על הטור salary:

```
CREATE CLUSTERED INDEX ind_salary ON dbo.Tiny_employees(salary);
```

שוב, הבה נחקור את תוכנית השאילתה, המוצגת בתרשים 2–14, עבור אותה שאילתה:

```
UPDATE dbo.Tiny_employees
  SET salary = salary * 1.1
WHERE salary < 25000;
```

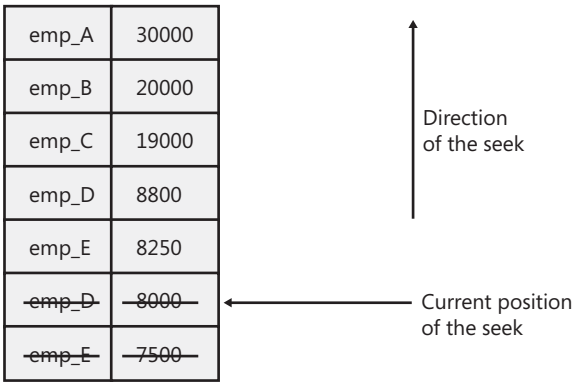
תרשים 2–14: תוכנית עבודה עבור המשפט UPDATE לאחר יצירת אינדקס ind_salary



שוב, לא קיימת שום צומת עדכון עבור האינדקס ind_name, מכיוון שזו תוכנית תחזוקה לפי–שורה. עם זאת, קיים אופרטור Table Spool. אסביר מדוע.

ה– clustered index seek סורק את השורות לפי הסדר של ערכי מפתחות ה–clustered. יש לנו את האינדקס–clustered ממוין על הטור salary, וזהו אותו טור שעובר עדכון בשאילתה שלנו. הבה נניח ש– SQL Server מחפש באינדקס–clustered מהערך הקטן ביותר לזה הגדול ביותר. הערך הראשון שיימצא בטבלה שלנו הוא 7500 עבור העובד emp_E. אנו מגדילים אותו ב–10 אחוז ל–8250. לפי חישוב זה, הרשומה תתחלף בין emp_D לבין emp_C עם משכורות "ישנות" של 8000 ו–19,000 בהתאמה. בשלב הבא, נמצא משכורת של 8000, ונגדיל אותה ל–8800. תרשים 2–15 מציג את העדכון במהלך ריצה.

תרשים 2–15: עדכון במהלך ריצה



אם החיפוש ממשיך, הוא יגיע לרשומה emp_E שוב ויעדכן אותה בפעם השנייה, דבר זה, כמובן, יהיה שגוי. כעת בחן את המיקום של Table Spool בתוכנית השאילתה הקודמת – היא מפרידה בין האופרטורים Clustered Index Seek ו– Clustered Index Update. הטבלה הזמנית קולטת את כל הרשומות מה– index seek בטרם היא ממשיכה עם העדכונים מול אותו אינדקס–clustered. הטבלה הזמנית, לא האינדקס, מייצרת את הערכים המעודכנים. לפיכך, הטבלה הזמנית מונעת עדכון של אותה רשומה פעמיים ומבטיחה תוצאה נכונה. בתוכנית שאילתות ה–IUD שלנו אנו קוראים לטבלה זמנית כזו Halloween spool.

אתה בטח תוהה מה המשותף בין Halloween לבין תוכנית השאילתה שזה עתה תוארה? עלינו לחזור אחורה כמעט 30 שנה בהיסטוריה של טכנולוגיית מסדי נתונים – ובפרט של אופטימיזציה של שאילתות – כדי לענות על שאלה זו. חוקרים במרכז המחקר Almaden בקליפורניה נתקלו באותה בעיה כאשר ניסו לגרום לאב הטיפוס של ה– query optimizer שלהם להשתמש באינדקסים בתוכניות עדכון. למעשה, הם השתמשו בטור salary ובשאילתת עדכון להגדלת משכורת בדומה למה שזה עתה עשיתי. יתרה מכך, להפתעתם הגדולה אף אחד לא קיבל משכורת נמוכה מ–25,000 לאחר שהריצו את שאילתת העדכון! היה זה חג Halloween של שנת 1977 (או אולי 1976?) כאשר החוקרים תיארו ודנו בתקלה. מאז, טקסטים, מסמכים ומאמרים רבים בנושא מסדי נתונים השתמשו במונח "Halloween" בשביל לכנות את בעיית העדכון הכפול שזה עתה הצגתי.

סיכום

עיבוד פיסי של שאילתות מורכב משני צעדים בסיסיים: קומפילציה של שאילתה והרצה של שאילתה. הקשר העיקרי בין שני הצעדים הוא תוכנית השאילתה. הבנה כיצד SQL Server מייצר את תוכניות השאילתה וכיצד הוא משתמש בתוכניות כדי לספק את תוצאת השאילתה חיונית למפתחי יישומים, מעצבי מסדי נתונים ומנהלי מערכת. שימוש יעיל בידע כיצד SQL Server מבצע עיבוד פיסי של שאילתה עשוי לשפר את הביצועים של שרת מסד הנתונים ואת זמן התגובה והתפוקה של היישום.

תודות

בעת שעבדתי על פרק זה קיבלתי עזרה רבה ממספר קולגות במיקרוסופט: מייק בלסזקזאק (Mike Blaszczak), אלקסיס בוקוואלאס (Alexis Boukouvalas), מילינד ג'וּשי (Milind Joshi), סטפנו סטפני (Stefano Stefani), דון וילן (Don Vilen), אומאצ'נדר ג'איאצ'אנדראן (Umachandar Jayachandran) ויוג'ין זבוקריצקי (Eugene Zabokritski) קראו והעירו על הפרק או חלק ממנו והעבירו לי משוב עשיר ורעיונות לשיפור התוכן.

3

כוונון שאילתות

פרק זה מניח את הבסיס לידע כוונון שאילתות הנדרש הן לספר זה והן לספר Microsoft SQL Server 2005: T-SQL Programming. (לשם הקיצור, אקרא לספר התכנות **T-SQL תכנות**, ואתייחס לשני הספרים כ"ספרים אלו"). כאן תוצג לפניך מתודולוגיית כוונון, תרכוש כלים לכוונון שאילתות, תלמד כיצד לנתח תוכניות עבודה ולבצע כוונון אינדקסים, ותלמד את המשמעות של הכנת נתוני דוגמה טובים ואת החשיבות של שימוש בפתרונות מבוססים–סטים.

כאשר בניתי את תוכן העניינים לספר זה, נתקלתי בדילמה של ממש בנוגע לפרק על כוונון שאילתות, דילמה בפניה אני עומד גם כאשר אני מלמד שפת T-SQL מתקדמת – האם רצוי שחומר זה יופיע בשלב מוקדם או מאוחר? מצד אחד, הפרק מספק מידע רקע חשוב הנדרש ליתר הספר; מצד שני, כמה מהשיטות המשמשות לכוונון שאילתות מערבות שאילתות מתקדמות – מעין בעיה של ביצה ותרנגולת. החלטתי לשלב את הפרק בשלב מוקדם של הספר, אך כתבתי אותו כיחידה עצמאית הניתנת לשימוש כמקור מידע. ההמלצה שלי היא שתקרא פרק זה לפני יתר הספר, וכאשר שאילתה משתמשת בשיטות שאינך מכיר עדיין, תתמקד רק באלמנטים המושגיים המתוארים במלל. מספר שאילתות ישתמשו בשיטות המתוארות בהמשך הספר (למשל, סיבוב על ציר, צבירות נעות, הפסוקית OVER, CUBE, CTEs וכו') או בספר **תכנות T-SQL** (למשל, טבלאות זמניות, סמנים, פרוצדורות, שילוב CLR, קומפילציה וכו'). אל תדאג אם השיטות אינן בהירות לך. עם זאת, תרגיש חופשי לקפוץ לפרק הרלוונטי אם אתה סקרן לגבי שיטה מסוימת. כאשר תסיים לקרוא את הספרים, אני מציע שתחזור לפרק זה ותסקור שנית כל שאילתה שהייתה לא מובנת בקריאה ראשונה כדי לוודא שאתה מבין היטב כיצד היא עובדת.

בסיום פרק זה, אספק מקורות בהם תוכל למצוא מידע נוסף בנושא.

נתוני דוגמה עבור פרק זה

לאורך הפרק, אשתמש בדוגמאות שלי במסד הנתונים Performance ובטבלאותיו. הרץ את הקוד בקטע–קוד 1–3 כדי ליצור את מסד הנתונים ואת טבלאותיו ולהכניס בהם נתוני דוגמה. שים לב שייקח לקוד מספר דקות לסיים.

קטע-קוד 1-3: קוד יצירת מסד נתונים וטבלאות לדוגמה

```
SET NOCOUNT ON;
USE master;
GO
IF DB_ID('Performance') IS NULL
  CREATE DATABASE Performance;
GO
USE Performance;
GO

-- Creating and Populating the Nums Auxiliary Table
IF OBJECT_ID('dbo.Nums') IS NOT NULL
  DROP TABLE dbo.Nums;
GO
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);
DECLARE @max AS INT, @rc AS INT;
SET @max = 1000000;
SET @rc = 1;

INSERT INTO Nums VALUES(1);
WHILE @rc * 2 <= @max
BEGIN
  INSERT INTO dbo.Nums SELECT n + @rc FROM dbo.Nums;
  SET @rc = @rc * 2;
END

INSERT INTO dbo.Nums
  SELECT n + @rc FROM dbo.Nums WHERE n + @rc <= @max;
GO

-- Drop Data Tables if Exist
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
GO
IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
GO
IF OBJECT_ID('dbo.Employees') IS NOT NULL
  DROP TABLE dbo.Employees;
GO
IF OBJECT_ID('dbo.Shippers') IS NOT NULL
  DROP TABLE dbo.Shippers;
GO
```

```
-- Data Distribution Settings
 DECLARE
   @numorders   AS INT,
   @numcusts    AS INT,
   @numemps     AS INT,
   @numshippers AS INT,
   @numyears    AS INT,
   @startdate   AS DATETIME;

 SELECT
   @numorders   =   1000000,
   @numcusts    =     20000,
   @numemps     =       500,
   @numshippers =         5,
   @numyears    =         4,
   @startdate   = '20030101';

-- Creating and Populating the Customers Table
 CREATE TABLE dbo.Customers
 (
   custid   CHAR(11)     NOT NULL,
   custname NVARCHAR(50) NOT NULL
 );

 INSERT INTO dbo.Customers(custid, custname)
   SELECT
     'C' + RIGHT('000000000' + CAST(n AS VARCHAR(10)), 10) AS custid,
     N'Cust_' + CAST(n AS VARCHAR(10)) AS custname
   FROM dbo.Nums
   WHERE n <= @numcusts;

 ALTER TABLE dbo.Customers ADD
   CONSTRAINT PK_Customers PRIMARY KEY(custid);

-- Creating and Populating the Employees Table
 CREATE TABLE dbo.Employees
 (
   empid    INT           NOT NULL,
   firstname NVARCHAR(25)  NOT NULL,
   lastname NVARCHAR(25)  NOT NULL
 );
```

```
INSERT INTO dbo.Employees(empid, firstname, lastname)
  SELECT n AS empid,
    N'Fname_' + CAST(n AS NVARCHAR(10)) AS firstname,
    N'Lname_' + CAST(n AS NVARCHAR(10)) AS lastname
  FROM dbo.Nums
  WHERE n <= @numemps;

ALTER TABLE dbo.Employees ADD
  CONSTRAINT PK_Employees PRIMARY KEY(empid);

-- Creating and Populating the Shippers Table
CREATE TABLE dbo.Shippers
(
  shipperid   VARCHAR(5)   NOT NULL,
  shippername NVARCHAR(50) NOT NULL
);

INSERT INTO dbo.Shippers(shipperid, shippername)
  SELECT shipperid, N'Shipper_' + shipperid AS shippername
  FROM (SELECT CHAR(ASCII('A') - 2 + 2 * n) AS shipperid
        FROM dbo.Nums
        WHERE n <= @numshippers) AS D;

ALTER TABLE dbo.Shippers ADD
  CONSTRAINT PK_Shippers PRIMARY KEY(shipperid);

-- Creating and Populating the Orders Table
CREATE TABLE dbo.Orders
(
  orderid   INT        NOT NULL,
  custid    CHAR(11)   NOT NULL,
  empid     INT        NOT NULL,
  shipperid VARCHAR(5) NOT NULL,
  orderdate DATETIME   NOT NULL,
  filler    CHAR(155)  NOT NULL DEFAULT('a')
);

INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  SELECT n AS orderid,
    'C' + RIGHT('000000000'
            + CAST(
                1 + ABS(CHECKSUM(NEWID())) % @numcusts
```

```
                AS VARCHAR(10)), 10) AS custid,
    1 + ABS(CHECKSUM(NEWID())) % @numemps AS empid,
    CHAR(ASCII('A') - 2
           + 2 * (1 + ABS(CHECKSUM(NEWID())) % @numshippers)) AS shipperid,
      DATEADD(day, n / (@numorders / (@numyears * 365.25)), @startdate)
        -- late arrival with earlier date
        - CASE WHEN n % 10 = 0
            THEN THEN 1 + ABS(CHECKSUM(NEWID())) % 30
            ELSE 0
          END AS orderdate
  FROM dbo.Nums
  WHERE n <= @numorders
  ORDER BY CHECKSUM(NEWID());

CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);

CREATE NONCLUSTERED INDEX idx_nc_sid_od_cid
  ON dbo.Orders(shipperid, orderdate, custid);

CREATE UNIQUE INDEX idx_unc_od_oid_i_cid_eid
  ON dbo.Orders(orderdate, orderid)
  INCLUDE(custid, empid);

ALTER TABLE dbo.Orders ADD
  CONSTRAINT PK_Orders PRIMARY KEY NONCLUSTERED(orderid),
  CONSTRAINT FK_Orders_Customers
    FOREIGN KEY(custid)     REFERENCES dbo.Customers(custid),
  CONSTRAINT FK_Orders_Employees
    FOREIGN KEY(empid)      REFERENCES dbo.Employees(empid),
  CONSTRAINT FK_Orders_Shippers
    FOREIGN KEY(shipperid)  REFERENCES dbo.Shippers(shipperid);
```

טבלת Orders היא טבלת הנתונים המרכזית ויש בה 1,000,000 הזמנות על פני ארבע שנים החל משנת 2003. בטבלת Customers יש 20,000 לקוחות, בטבלת Employees 500 עובדים, ובטבלת Shippers 5 מובילים. שים לב שפיזרתי את תאריכי ההזמנה, קודי הלקוחות, קודי העובדים וקודי המובילים בטבלת Orders על ידי שימוש בפונקציות היוצרות ערכים אקראיים. ייתכן שלא תקבל חזרה מהשאילתות מספר שורות זהה לזה שאני אקבל, אך סטטיסטית הן צריכות להיות קרובות למדי.

טבלת Nums היא טבלת עזר של מספרים, המכילה טור אחד בלבד הנקרא n, המכיל מספרים שלמים בתחום מ–1 ועד 1,000,000.

הקוד בקטע–קוד 1–3 יוצר את האינדקסים הבאים על טבלת Orders:

- **idx_cl_od** – אינדקס–clustered על orderdate.
- **PK_Orders** – אינדקס–nonclustered ייחודי על orderid, הנוצר בצורה עקיפה על ידי המפתח הראשי.
- **idx_nc_sid_od_cid** – אינדקס–nonclustered על (shipperid, orderdate ,custid).
- **idx_unc_od_oid_i_cid_eid** – אינדקס–unique nonclustered על טורי המפתח (orderdate, orderid), וטורים מוכללים שאינם מפתח (included nonkey) – (custid, empid).

מבני אינדקסים ותכונותיהם יוסברו בהמשך בסעיף "כוונון אינדקסים".

מתודולוגיית כוונון

סעיף זה מתאר מתודולוגיית כוונון שפותחה בחברה בה אני עובד – Solid Quality Learning – והוטמעה אצל לקוחותינו. יש לתת את הקרדיט למנטורים בחברה שלקחו חלק בפיתוח והטמעת המתודולוגיה, במיוחד לאנדרו ג. קלי (Andrew J. Kelly), בריאן מוראן (Brian Moran), פרננדו ג. גוורו (Fernando G. Guerrero), אלאדיו רינקון (Eladio Rincón), דיין סרקה (Dejan Sarka), מייק הוטק (Mike Hotek) ורון טלמג' (Ron Talmage), אם נזכיר רק חלק מהשמות.

ובכן, כאשר המערכת שלך סובלת מבעיות ביצועים, כיצד אתה מתחיל לפתור את הבעיה?

התשובה לשאלה זו מזכירה לי תוכניתן ומנהל מערכות מידע בחברה בה עבדתי לפני שנים. התוכניתן נדרש לסיים לכתוב תוכנית ולהטמיע אותה, אך בקוד שלו היה באג אותו לא הצליח למצוא. הוא הפיק תדפיס של הקוד (שהיה עבה למדי) וניגש למנהל מערכות המידע, שהיה בפגישה. המנהל היה מומחה באבחון באגים, זו הסיבה שהתוכניתן רצה להיוועץ בו. המנהל לקח את התדפיס העבה, פתח אותו, ומייד הצביע לשורה מסוימת בקוד. "הנה הבאג שלך" הוא אמר, "עכשיו לך". לאחר שהפגישה הסתיימה, התוכניתן שאל את המנהל כיצד הוא מצא את הבאג כל–כך מהר? המנהל השיב, "ידעתי שבכל מקום שאצביע יהיה באג".

חזרה לכוונון שאילתות, ניתן להצביע על כל מקום במסד הנתונים ויהיה מקום לבצע שם כוונון. השאלה היא, האם זה כדאי? למשל, האם משתלם לכוונן אספקטים של עבודה בריבוי משתמשים במקביל (concurrency) במערכת, אם סך כל החסימות (blocking) תורם לאחוז אחד בלבד של ההמתנות במערכת? חשוב לעקוב אחר מסלול או מתודולוגיה המובילים אותך דרך סדרת צעדים לאזורי הבעיה או לצווארי הבקבוק העיקריים במערכת – אלו התורמים למרבית ההמתנות. סעיף זה יציג מתודולוגיה כזו.

בטרם תמשיך, הסר את האינדקס–clustered הקיים מטבלת Orders:

```
USE Performance;
GO
DROP INDEX dbo.Orders.idx_cl_od;
```

נניח שהמערכת שלך כולה סובלת מבעיות ביצועים – משתמשים מתלוננים ש"הכל איטי". קטע–קוד 2–3 מכיל דגימה של שאילתות הרצות באופן קבוע במערכת שלך.

קטע–קוד 2–3: שאילתות לדוגמה

```
SET NOCOUNT ON;
USE Performance;
GO

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 3;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 5;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 7;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate = '20060212';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate = '20060118';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate = '20060828';
```

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060101'
  AND orderdate < '20060201';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060401'
  AND orderdate < '20060501';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20070301';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060501'
  AND orderdate < '20060601';
```

אתחל מחדש את מופע ה– SQL Server שלך, ואז הרץ את הקוד בקטע–קוד 2–3 מספר פעמים (נאמר, 10). SQL Server ישמור פנימית מידע ביצועים עליו תסתמך בהמשך. אתחול מחדש של המופע שלך יאפס כמה מהמונים עליהם תסתמך בהמשך.

כאשר מתמודדים עם בעיות ביצועים, מקצועני מסדי נתונים נוטים להתמקד באספקטים הטכניים של המערכת, כגון תורי משאבים, ניצול משאבים, וכו'. אך משתמשים תופסים בעיות ביצועים פשוט כהמתנות – הם שולחים בקשה ונאלצים להמתין לקבלת התוצאה. תשובה המגיעה למעלה משלוש שניות, לאחר שבוצעה בקשה אינטראקטיבית, נתפסת לרוב על ידי משתמשים כבעיית ביצועים. לא ממש מעניין אותם כמה פקודות ממתינות בממוצע בכל דיסק או מהו אחוז הגישה ל–cache (cache hit ratio), ולא מעניינות אותם חסימות, ניצול מעבדים, תוחלת חיים ממוצעת של דף ב–cache וכו'. מה שמעניין אותם זה המתנות, וזהו המקום בו כוונון שאילתות צריך להתחיל.

מתודולוגיית הכוונון עליה אני ממליץ מיישמת גישת מלמעלה–למטה. היא מתחילה בחקירת המתנות ברמת המופע (instance) של SQL Server, וממשיכה מטה דרך סדרת צעדים עד שהתהליך/הרכיבים המייצרים את עיקר ההמתנות במערכת מזוהים. לאחר שזיהית את התהליכים הבעייתיים, באפשרותך להתמקד בכוונונם.

להלן השלבים העיקריים של המתודולוגיה:

1. ניתוח המתנות ברמת המופע.

2. קשר בין המתנות לתורים.
3. קבע דרך פעולה.
4. המשך מטה לרמת מסד הנתונים/הקובץ.
5. המשך מטה לרמת התהליך.
6. כוונן אינדקסים/שאילתות.

הסעיפים הבאים מתארים כל שלב בפירוט.

ניתוח המתנות ברמת המופע (instance)

השלב הראשון במתודולוגיית הכוונון הוא לזהות, ברמת המופע של SQL Server, לאילו סוגי המתנות יש את התרומה הרבה ביותר להמתנות במערכת. ב– SQL Server 2005, אתה עושה זאת על ידי ביצוע שאילתה מול DMV (Dynamic Management View) הנקרא sys.dm_os_wait_stats; ב– SQL Server 2000, אתה עושה זאת על ידי הרצת הפקודה DBCC SQLPERF(WAITSTATS). ל–DMV שזה עתה הזכרתי ב– SQL Server 2005 קיים תיעוד מלא, ואני מעודד אותך לקרוא את הסעיף המתאר אותו ב– Books Online. איני יודע את הסיבה, אך הפקודה ב– SQL Server 2000 אינה מתועדת והיא הופיעה רק כמה שנים לאחר שהמוצר יצא לשוק. בכל אופן, מהתיעוד ב– SQL Server 2005 ניתן ללמוד על הסוגים השונים של המתנות הרלוונטיות גם ל– SQL Server 2000.

ה– sys.dm_os_wait_stats DMV מכיל 194 סוגי המתנות, בעוד שהפקודה ב– SQL Server 2000 תחזיר 77. אם תחשוב על כך, אלו מספרים קטנים וניתנים לטיפול הנוחים לעבודה כנקודת פתיחה. מספר כלי ביצועים אחרים נותנים לך יותר מדי מידע בהתחלה, ויוצרים מצב בו מרוב עצים לא רואים את היער. אמשיך את הדיון בהנחה שאתה עובד ב– SQL Server 2005.

הרץ את השאילתה הבאה כדי להחזיר את ההמתנות במערכת שלך ממוינות לפי סוג:

```
SELECT
  wait_type,
  waiting_tasks_count,
  wait_time_ms,
  max_wait_time_ms,
  signal_wait_time_ms
FROM sys.dm_os_wait_stats
ORDER BY wait_type;
```

טבלה 1–3 מציגה גרסה מקוצרת של התוצאות שקיבלתי כאשר הרצתי שאילתה זו במערכת שלי.

טבלה 1–3: תוכן של sys.dm_os_wait_stats בצורה מקוצרת

wait_type	*waiting_ tasks_count*	*wait_ time_ms*	*max_wait_ time_ms*	*signal_wait_ time_ms*
…				
ASYNC_IO_ COMPLETION	9	43993	40247	20
ASYNC_ NETWORK_IO	69	1682	941	110
CHKPT	1	3234	3234	0
IO_COMPLETION	29167	466270	620	1932
LATCH_EX	41	13589	2082	290
LATCH_SH	3	110	110	30
LATCH_UP	0	0	0	0
LAZYWRITER_SLEEP	284462	266925468	18236	48329
LCK_M_S	3	1882	1612	10
LCK_M_U	41	26898	1992	0
LCK_M_X	0	0	0	0
LOGBUFFER	9495	194920	751	1692
LOGMGR_ RESERVE_APPEND	200	198745	1331	50
OLEDB	1451	3034	851	0
PAGELATCH_EX	123	60	20	40
PAGELATCH_SH	25300	365115	2012	54288
PAGELATCH_UP	29	7080	1041	0
PAGEIOLATCH_EX	1259	46536	721	480
PAGEIOLATCH_SH	39491	358205	1762	2733
PAGEIOLATCH_UP	382	6389	480	70
RESOURCE_QUEUE	0	0	0	0
RESOURCE_ SEMAPHORE	0	0	0	0
RESOURCE_ SEMAPHORE_MUTEX	0	0	0	0

wait_type	*waiting_ tasks_count*	*wait_ time_ms*	*max_wait_ time_ms*	*signal_wait_ time_ms*
RESOURCE_ SEMAPHORE_ QUERY_COMPILE	0	0	0	0
RESOURCE_ SEMAPHORE_ SMALL_QUERY	0	0	0	0
WRITELOG	5176	97400	781	3114
…				

שים לב: כמובן שלא רצוי להסיק מסקנות לגבי מערכות תפעוליות מהפלט שאני השגתי. אין צורך לציין שהמכונה הפרטית שלי, או מכונת הבדיקות או סביבת הבדיקות האישית שלך, לא בהכרח ישקפו מערכת תפעולית אמיתית. אני משתמש בפלט שבטבלה זו רק למטרות הדגמה. בהמשך אציין אילו סוגי המתנות הן לרוב הנפוצות ביותר במערכות תפעוליות.

ה–DMV צובר ערכים מאז שהשרת אותחל בפעם האחרונה. אם ברצונך לאפס את הערכים שלו, הרץ את הקוד הבא (אך אל תריץ אותו כעת):

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
```

ב– SQL Server 2000, קוד זה יאפס את סטטיסטיקות ההמתנה:

```
DBCC SQLPERF(WAITSTATS, CLEAR);
```

ה– DMV sys.dm_os_wait_stats מכיל את הטורים הבאים: wait_type; waiting_tasks_count, שהוא מספר ההמתנות בסוג המתנה זה; wait_time_ms, שהוא סך זמן ההמתנה לסוג המתנה זה באלפיות–שנייה (כולל signal_wait_time); max_wait_time_ms; ו– signal_wait_time, שהוא ההפרש בין הזמן שה–thread הממתין קיבל את האות לבין הזמן בו הוא התחיל לרוץ.

כפי שציינתי קודם, תוכל למצוא מידע על סוגי ההמתנות השונים ב– Books Online. בסוף פרק זה, אכוון אותך למקורות המספקים מידע מפורט יותר לגבי סוגי ההמתנות השונים.

בין סוגי ההמתנות השונים, תמצא כאלו הקשורים בנעילות, latches (נעילות lightweight), I/O[1] (כולל I/O latches), transaction log, זיכרון, קומפילציות, OLEDB (linked servers) ורכיבי OLEDB אחרים), וכו'. בדרך–כלל, תרצה להתעלם מסוגים מסוימים של המתנות – למשל, אלו המערבות המתנה של thread (הכוונה היא שה–thread פשוט מושהה, ולא עושה כלום). ודא שאתה מסנן המתנות לא רלוונטיות כך שהן לא יעוותו את החישובים שלך.

1 I/O – קלט/פלט.

מניסיוני ב– Solid Quality Learning, המתנות I/O הן באופן מובהק הסוגים הנפוצים ביותר של המתנות עבורן לקוחותינו נזקקים לעזרה. קיימות מספר סיבות לכך. I/O הוא לרוב המשאב היקר ביותר המעורב בפעילויות של מניפולציית–נתונים. וכן, כאשר שאילתות או אינדקסים אינם מעוצבים ומכוונים היטב, התוצאה היא לרוב I/O עודף. כמו כן, כאשר לקוחות חושבים על מכונות "חזקות", הם לרוב ממקדים את תשומת הלב שלהם במעבד ובזיכרון, ואינם מקדישים תשומת לב מספיקה לתת–המערכת של ה–I/O. מערכות מסדי נתונים צריכות תת–מערכות I/O חזקות.

כמובן, אנו עוסקים גם בתחומים בעייתיים אחרים. ישנן מערכות שלא בהכרח ניגשות למנות גדולות של הנתונים; במקום זאת, מערכות אלו מערבות תהליכים הניגשים למנות קטנות של הנתונים לעיתים קרובות. זהו לרוב המקרה במערכות עיבוד טרנזקציות online (OLTP), אשר יש להן פרוצדורות מאוחסנות ושאילתות הניגשות למנות קטנות של הנתונים אך מורצות לעיתים מאוד קרובות. בסביבות כאלו, קומפילציות וקומפילציות–חוזרות של הקוד עשויות להיות הסיבה העיקרית לצוואר בקבוק. מערכות OLTP מערבות גם הרבה שינויי נתונים במנות קטנות, ולעיתים קרובות ה– transaction log הופך להיות צוואר בקבוק במערכות כאלו. מסד הנתונים tempdb עשוי גם הוא להיות צוואר בקבוק רציני משום שכל הטבלאות הזמניות, בין אם נוצרו בעקיפין על ידי תוכנית עבודה, ובין במישרין, נוצרות ב– tempdb. SQL Server משתמש גם בנפח של tempdb לביצוע פעילויות אחרות. לעיתים, אנו מוצאים גם מערכות עם בעיות הקשורות לעבודה של ריבוי משתמשים במקביל (חסימות), וכן בעיות אחרות.

הבה נשוב למידע ההמתנות שקיבלת מה–DMV. סביר שלא יהיה לך נוח לעיין בכל סוגי ההמתנות ולגלות ידנית אילו המשמעותיות ביותר. תרצה לבודד את ההמתנות הארוכות – אלו שבסך–הכל מצטברות לאחוז סף מסוים מכלל ההמתנות במערכת. תוכל להשתמש במספרים כמו 80 אחוז או 90 אחוז, משום שלרוב מספר קטן של סוגי המתנות תורם לעיקר ההמתנות במערכת.

השאילתה הבאה מבודדת את ההמתנות הארוכות ביותר המצטברות בסך–הכל ל–90 אחוז מזמן ההמתנה במערכת, והיא מפיקה (במערכת שלי) את הפלט המוצג בטבלה 2–3:

```
WITH Waits AS
(
  SELECT
    wait_type,
    wait_time_ms / 1000. AS wait_time_s,
    100. * wait_time_ms / SUM(wait_time_ms) OVER() AS pct,
    ROW_NUMBER() OVER(ORDER BY wait_time_ms DESC) AS rn
  FROM sys.dm_os_wait_stats
  WHERE wait_type NOT LIKE '%SLEEP%'
  -- filter out additional irrelevant waits
)
```

```
SELECT
  W1.wait_type,
  CAST(W1.wait_time_s AS DECIMAL(12, 2)) AS wait_time_s,
  CAST(W1.pct AS DECIMAL(12, 2)) AS pct,
  CAST(SUM(W2.pct) AS DECIMAL(12, 2)) AS running_pct
FROM Waits AS W1
  JOIN Waits AS W2
    ON W2.rn <= W1.rn
GROUP BY W1.rn, W1.wait_type, W1.wait_time_s, W1.pct
HAVING SUM(W2.pct) - W1.pct < 90 -- percentage threshold
ORDER BY W1.rn;
```

טבלה 2–3: ההמתנות הארוכות ביותר

wait_type	*wait_time_s*	*pct*	*running_pct*
IO_COMPLETION	466.24	23.98	23.98
PAGEIOLATCH_SH	365.08	18.78	42.76
ASYNC_NETWORK_IO	358.21	18.42	61.18
LOGMGR_RESERVE_APPEND	198.75	10.22	71.40
LOGBUFFER	194.92	10.02	81.42
WRITELOG	97.40	5.01	86.43
PAGEIOLATCH_EX	46.54	2.39	88.83
ASYNC_IO_COMPLETION	43.99	2.26	91.09

שאילתה זו משתמשת בשיטות לחישוב צבירות נעות, אותן אסביר בשלב מאוחר יותר בספר. זכור, התמקד כעת במושגים ולא בשיטות ששימשו להשגתם. שאילתה זו מחזירה את ההמתנות הארוכות ביותר המצטברות ל–90 אחוז מההמתנות במערכת, לאחר שסיננו החוצה סוגי המתנות לא רלוונטיים. כמובן, תוכל לשנות את רמת הסף ולסנן סוגי המתנות אחרות שאינן רלוונטיות במערכת שלך. אם ברצונך לראות לפחות n שורות בפלט (נאמר n=10), הוסף את הביטוי OR W1.rn <= 10 לפסוקית HAVING. לכל סוג המתנה, השאילתה מחזירה את המידע הבא: סך זמן ההמתנה בשניות שתהליכים המתינו בזמן המתנה זה מאז שהמערכת אותחלה לאחרונה או שהמונים אופסו; אחוז זמן ההמתנה של סוג זה מסך–הכל; והאחוז הנע מסוג ההמתנה הגבוהה ביותר ועד לנוכחי.

שים לב: ב– sys.dm_os_wait_stats DMV, הטור wait_time_ms מייצג את סך זמן ההמתנה של כל התהליכים שהמתינו בסוג זה, אפילו אם מספר תהליכים המתינו בו–זמנית. עדיין, מספרים אלו יתנו לך לרוב תחושה נכונה של תחומי הבעיה העיקריים במערכת.

כאשר תבחן את ההמתנות הגבוהות ביותר המוצגות בטבלה 2–3, תוכל לזהות שלושה תחומי בעיה: I/O, רשת ו– transaction log. עם מידע זה ביד אתה מוכן לשלב הבא.

אני מוצא ששימושי מאוד גם לאסוף מידע המתנה בטבלה ולעדכן אותה בפרקי זמן קבועים (למשל, פעם בשעה). על ידי כך תוכל לנתח את הפיזור של ההמתנות במהלך היום ולזהות זמני עומס.

הרץ את הקוד הבא כדי ליצור את טבלת WaitStats:

```
USE Performance;
GO
IF OBJECT_ID('dbo.WaitStats') IS NOT NULL
  DROP TABLE dbo.WaitStats;
GO

SELECT GETDATE() AS dt,
  wait_type, waiting_tasks_count, wait_time_ms,
  max_wait_time_ms, signal_wait_time_ms
INTO dbo.WaitStats
FROM sys.dm_os_wait_stats
WHERE 1 = 2;

ALTER TABLE dbo.WaitStats
  ADD CONSTRAINT PK_WaitStats PRIMARY KEY(dt, wait_type);
CREATE INDEX idx_type_dt ON dbo.WaitStats(wait_type, dt);
```

הגדר עיבוד מתוזמן (scheduled job) אשר רץ בפרקי זמן קבועים ומשתמש בקוד הבא לטעינת הנתונים הנוכחיים מה–DMV:

```
INSERT INTO Performance.dbo.WaitStats
  SELECT GETDATE(),
    wait_type, waiting_tasks_count, wait_time_ms,
    max_wait_time_ms, signal_wait_time_ms
FROM sys.dm_os_wait_stats;
```

זכור שמידע ההמתנה ב–DMV הוא מצטבר. כדי להשיג את ההמתנות שהתרחשו בכל פרק זמן עליך להפעיל self-join בין שני מופעים של הטבלה – אחד המייצג את הדגימות הנוכחיות, והאחר מייצג את הדגימות הקודמות. תנאי ה–join יתאים לכל שורה נוכחית את השורה המייצגת את הדגימה הקודמת לאותו סוג המתנה. אז תוכל להחסיר את זמן ההמתנה המצטבר של הדגימה הקודמת מזה הנוכחי, וכך לייצר את זמן ההמתנה במהלך פרק הזמן. הקוד הבא יוצר את הפונקציה fn_interval_waits, המיישמת לוגיקה זו:

```
IF OBJECT_ID('dbo.fn_interval_waits') IS NOT NULL
  DROP FUNCTION dbo.fn_interval_waits;
GO

CREATE FUNCTION dbo.fn_interval_waits
  (@fromdt AS DATETIME, @todt AS DATETIME)
RETURNS TABLE
AS

RETURN
  WITH Waits AS
  (
    SELECT dt, wait_type, wait_time_ms,
      ROW_NUMBER() OVER(PARTITION BY wait_type
                        ORDER BY dt) AS rn
    FROM dbo.WaitStats
    WHERE dt >= @fromdt
      AND dt < @todt + 1
  )
  SELECT Prv.wait_type, Prv.dt AS start_time,
    CAST((Cur.wait_time_ms - Prv.wait_time_ms)
           / 1000. AS DECIMAL(12, 2)) AS interval_wait_s
  FROM Waits AS Cur
    JOIN Waits AS Prv
      ON Cur.wait_type = Prv.wait_type
      AND Cur.rn = Prv.rn + 1
      AND Prv.dt <= @todt;
GO
```

הפונקציה מקבלת את הגבולות של תקופה אותה ברצונך לנתח. למשל, השאילתה הבאה מחזירה את ההמתנות בפרק הזמן של התקופה '20060212' ועד '20060215' (כולל), ממוינות לפי סך–הכל של כל סוג המתנה בסדר יורד, סוג המתנה וזמן התחלה:

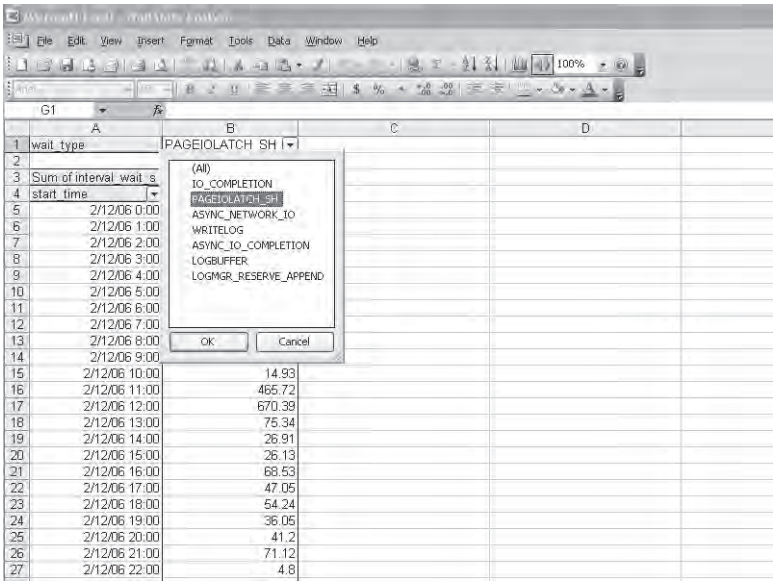
```
SELECT wait_type, start_time, interval_wait_s
FROM dbo.fn_interval_waits('20060212', '20060215') AS F
ORDER BY SUM(interval_wait_s) OVER(PARTITION BY wait_type) DESC,
  wait_type, start_time;
```

אני מוצא שטבלאות pivot של Microsoft Office Excel או קוביות Analysis Services שימושיות מאוד לניתוח מידע כזה בצורה גרפית. כלים אלו מאפשרים לך לראות בקלות את הפיזור של ההמתנות בצורה גרפית. למשל, נניח שברצונך לנתח את ההמתנות עבור התקופה '20060212' ועד '20060215' על ידי שימוש בטבלאות pivot של Excel. הכן את ה– VIntervalWaits view הבא, שישמש כמקור הנתונים החיצוני לטבלת ה–pivot:

```
IF OBJECT_ID('dbo.VIntervalWaits') IS NOT NULL
  DROP VIEW dbo.VIntervalWaits;
GO

CREATE VIEW dbo.VIntervalWaits
AS

SELECT wait_type, start_time, interval_wait_s
FROM dbo.fn_interval_waits('20060212', '20060215') AS F;
GO
```

צור טבלת pivot ותרשים pivot ב–Excel, וציין את ה– VIntervalWaits view כמקור הנתונים החיצוני של טבלת ה–pivot. תרשים 1–3 מראה כיצד נראית טבלת ה–pivot עם נתוני הדוגמה שלי, לאחר סינון של ההמתנות הארוכות ביותר בלבד.

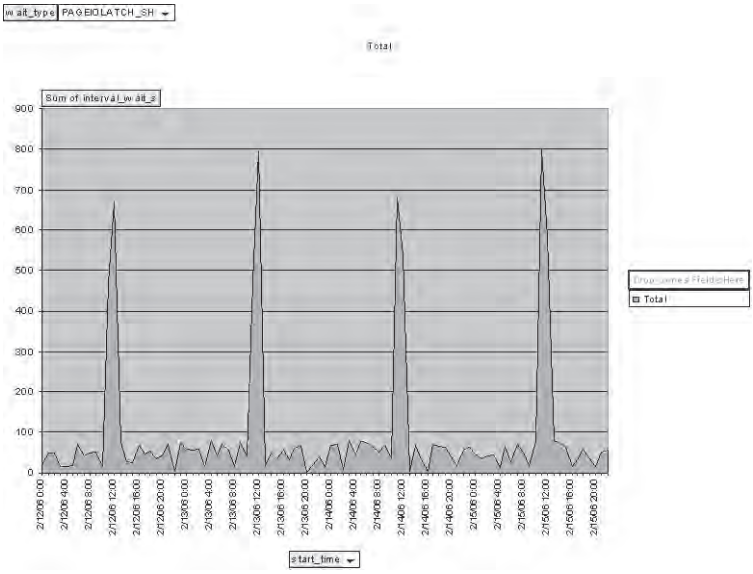
תרשים 1–3: טבלת pivot ב–Excel



בתרשים 2–3 מופיע תרשים pivot, המציג גרפית את הפיזור של סוג ההמתנה PAGEIOLATCH_SH על פני תקופת הקלט.

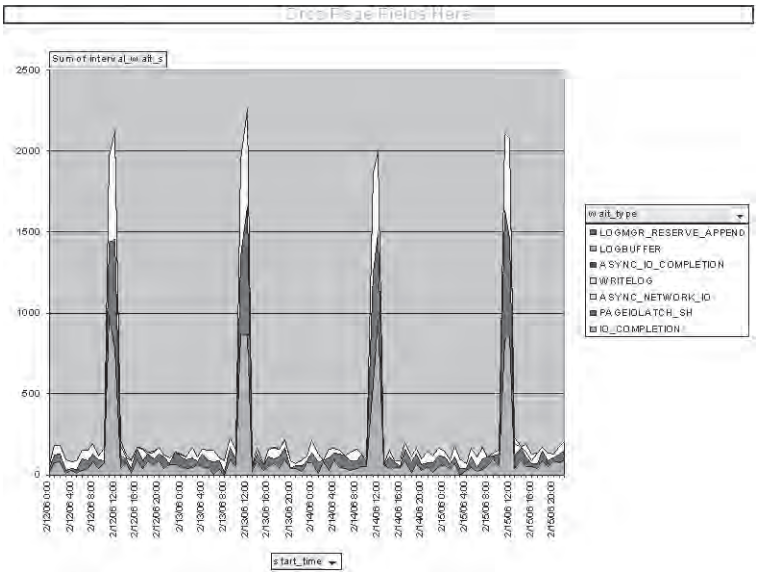
סוג ההמתנה PAGEIOLATCH_SH מציין המתנות I/O עבור פעולות קריאה. ניתן לראות בקלות שבמקרה שלנו, קיימים שיאים בולטים בכל יום בסביבות 12:00.

תרשים 3–2: תרשים pivot 1 ב–Excel



בתרשים 3–3 נראה תרשים pivot המציג בצורה גרפית את הפיזור של כל סוגי ההמתנות הארוכים ביותר.

תרשים 3–3: תרשים pivot 2 ב–Excel



שוב, ניתן לראות שמרבית ההמתנות קורות מדי יום בסביבות 12:00.

כדי להדגים עד כמה שימושי יכול להיות הניתוח של המתנות תקופתיות, אקח דוגמה מהשטח. באחד מפרויקטי הכוונון שלי מצאתי זמני עומס של I/O latches כל ארבע שעות שנמשכו לא מעט זמן (כמעט כל ארבע השעות). באופן טבעי, במקרה כזה אתה מחפש עיבודים הרצים בצורה מתוזמנת. ואכן, מהר מאוד בודד ה"עבריין". היה זה עיבוד מתוזמן שהפעיל את הפרוצדורה המאוחסנת sp_updatestats על כל מסד נתונים כל ארבע שעות ורץ במשך קרוב לארבע שעות. פרוצדורה מאוחסנת זו משמשת לעדכון סטטיסטיקות גלובלית ברמת מסד הנתונים. סטטיסטיקות הן היסטוגרמות המתוחזקות עבור טורים בהם משתמש ה–optimizer כדי לקבוע סלקטיביות של שאילתות, צפיפות של joins, וכו'. כפי הנראה, במקרה זה, כמה שנים קודם לכן לשאילתה מסוימת היו ביצועים לא טובים בגלל מחסור בסטטיסטיקות מעודכנות על טור אינדקס מסוים. הלקוח קיבל אז המלצה לרענן סטטיסטיקות, ונראה שהרצת הפרוצדורה המאוחסנת פתרה את הבעיה. מאז, הלקוח הריץ sp_updatestats גלובלית בכל ארבע שעות.

שים לב ש– SQL Server יוצר ומעדכן סטטיסטיקות אוטומטית. לרוב, התחזוקה האוטומטית של סטטיסטיקות מספיקה, ועליך להתערב ידנית רק במקרים מיוחדים. ואם הינך מתערב ידנית, אל תשתמש ב–sp_updatestats גלובלית! הפרוצדורה המאוחסנת sp_updatestats שימושית בעיקר לרענון סטטיסטיקות גלובלית לאחר שדרוג של המוצר, או לאחר חיבור מסד נתונים מגרסה קודמת של המוצר או של רמת ה– service pack. באופן אירוני, כשנמצאה הבעיה, השאילתה שהביאה ליצירת העיבוד אפילו לא הייתה עוד בשימוש במערכת. פשוט הסרנו את העיבוד ונתנו ל– SQL Server להשתמש בתחזוקה האוטומטית של סטטיסטיקות. כמובן שהגרף של ה– I/O latches פשוט השתטח ובעיית הביצועים נעלמה.

קישור בין המתנות לתורים

לאחר שזיהית את ההמתנות הארוכות ביותר ברמת המופע, עליך לקשר אותן לתורים כדי לזהות את המשאבים הבעייתיים. למשימה זו אתה משתמש בעיקר ב– System Monitor. למשל, אם זיהית בשלב הקודם המתנות הקשורות ב–I/O, תבדוק את תורי ה–I/O השונים, אחוז הגישה ל–cache ומוני זיכרון. בממוצע צריכות להיות פחות משתי פקודות I/O ממתינות בתור I/O בכל דיסק. אחוז הגישה ל–cache צריך להיות גבוה ככל הניתן.

באשר לזיכרון, הוא קשור ביותר ל–I/O משום שככל שיש לך יותר זיכרון, כך יכולים דפים (נתונים ותוכניות עבודה) להישאר יותר זמן ב–cache, וכך להפחית את הצורך ב–I/O פיסי. עם זאת, אם יש לך בעיות I/O כיצד תדע אם הוספת זיכרון אכן תעזור? עליך להכיר את הכלים שיסייעו לך לבצע את הבחירה הנכונה. למשל, המונה SQL Server: Buffer Manager – Page life expectancy יאמר לך כמה שניות בממוצע צפוי שדף יישאר ב–cache מבלי שפונים אליו. ערכים נמוכים מלמדים שהוספת זיכרון תאפשר לדפים להישאר ב–cache זמן ממושך יותר, בעוד שערכים גבוהים מלמדים שהוספת זיכרון לא תעזור הרבה במובן זה. המספרים בפועל תלויים בציפיות שלך ובתדירות בה אתה מריץ שאילתות הנשענות על אותם נתונים/תוכניות עבודה. לרוב, מספרים גבוהים מכמה מאות מעידים על מצב זיכרון טוב.

אך הבה נאמר שיש לך ערכים נמוכים מאוד במונה. האם המשמעות היא שעליך להוסיף זיכרון? הוספת זיכרון במקרה כזה סביר שתעזור, אך אולי ישנן שאילתות החסרות אינדקסים חשובים על טבלאות המקור אשר גורמות לביצוע I/O מופרז, דבר ממנו ניתן להימנע על ידי עיצוב אינדקסים טוב יותר. עם פחות I/O ופחות לחץ על הזיכרון, הבעיה יכולה להיפתר ללא השקעה בחומרה. כמובן שאם אתה ממשיך בניתוח ומגלה שהאינדקסים והשאילתות שלך מכוונים היטב, אז תשקול שדרוגי חומרה.

בדומה, אם זיהית סוגי המתנה אחרים כגבוהים ביותר, תבדוק את התורים הרלוונטיים ואת ניצול המשאבים. למשל, אם ההמתנות מערבות קומפילציות/קומפילציות–חוזרות, תבדוק את מוני הקומפילציות ואת מוני ניצול המעבד, וכו'.

SQL Server 2005 מספק לך DMV הנקרא sys.dm_os_performance_counters המכיל את כל מוני SQL Server הקשורים באובייקטים שתוכל למצוא ב– System Monitor. לרוע המזל, DMV זה לא נותן לך את המונים הכלליים יותר, כגון ניצול מעבדים, תורי I/O, וכו'. את אלו עליך לנתח חיצונית. למשל, כאשר הרצתי את השאילתה הבאה על המערכת שלי קיבלתי את הפלט המוצג (בצורה מקוצרת) בטבלה 3–3:

```
SELECT
  object_name,
  counter_name,
  instance_name,
  cntr_value,
  cntr_type
FROM sys.dm_os_performance_counters;
```

טבלה 3–3: תוכן של sys.dm_os_performance_counters בצורה מקוצרת

object_name	*counter_name*	*instance_name*	*cntr_value*	*cntr_type*
MSSQL$S2:Buffer Manager	Buffer cache hit ratio		634	537003264
MSSQL$S2:Buffer Manager	Buffer cache hit ratio base		649	1073939712
MSSQL$S2:Buffer Manager	Page lookups/sec		62882649	272696576
MSSQL$S2:Buffer Manager	Free list stalls/sec		22370	272696576
MSSQL$S2:Buffer Manager	Free pages		43	65792
MSSQL$S2:Buffer Manager	Total pages		21330	65792
MSSQL$S2:Buffer Manager	Target pages		44306	65792

object_name	counter_name	instance_name	cntr_value	cntr_type
MSSQL$S2:Buffer Manager	Database pages		19134	65792
MSSQL$S2:Buffer Manager	Reserved pages		0	65792
MSSQL$S2:Buffer Manager	Stolen pages		2153	65792
…				

שים לב שב– SQL Server 2000 תבצע במקום זאת שאילתה מול: master.dbo.sysperfinfo.

ייתכן שתמצא את היכולת לבצע שאילתות על מוני ביצועים אלו ב– SQL Server שימושית משום שבאפשרותך להשתמש במניפולציית–שאילתות בשביל לנתח את הנתונים. כמו עם מידע המתנות, באפשרותך לאסוף מוני ביצועים בטבלה בפרקי זמן קבועים, ואז להשתמש בשאילתות ובכלים כמו טבלאות pivot כדי לנתח את הנתונים על פני זמן.

קביעת דרך פעולה

נקודה זו – לאחר שזיהית את סוגי ההמתנות העיקריים ואת המשאבים המעורבים – היא צומת בתהליך הכוונון. בהתבסס על ממצאיך עד כה, תקבע דרך פעולה להמשך חקירה. במקרה שלנו, עלינו לזהות את הסיבות ל–I/O, להמתנות הקשורות ברשת, ולהמתנות הקשורות ב– transaction log; אז נמשיך במסלול המבוסס על הממצאים שלנו. אך אם הצעדים הקודמים זיהו בעיות של חסימות, בעיות קומפילציה/קומפילציה–חוזרת, או אחרות, יהיה עליך להמשיך בדרך פעולה שונה לחלוטין. כאן אדגים כוונון של בעיות ביצועים הקשורות ב–I/O, ובסוף הפרק אספק מקורות למידע נוסף על בעיות ביצועים מסוגים אחרים, כמה מהן מוצגות בספרים אלו.

ממשיכים מטה לרמת מסד הנתונים/הקובץ

השלב הבא בתהליך הכוונון שלנו הוא להמשיך מטה לרמת מסד הנתונים/הקובץ. ברצונך לבודד את מסדי הנתונים המעורבים במרבית העלות. בתוך מסד הנתונים, ברצונך להמשיך מטה לסוג הקובץ (נתונים/לוג), שכן דרך הפעולה בה תבחר תלויה בסוג הקובץ. הכלי שמאפשר לך לנתח מידע I/O ברמת מסד הנתונים/הקובץ הוא DMF (Dynamic Management Function) הנקרא sys.dm_io_virtual_file_stats. הפונקציה מקבלת קוד מסד נתונים וקוד קובץ כקלט, ומחזירה מידע I/O על קובץ מסד הנתונים של הקלט. כדי לבקש מידע על כל מסדי הנתונים ועל כל הקבצים אתה מספק NULLs בשני הפרמטרים של הפונקציה. שים לב שב– SQL Server 2000 אתה מבצע שאילתה על

פונקציה הנקראת: fn_virtualfilestats, ומספק לה 1– בשני הפרמטרים כדי לקבל מידע על כל מסדי הנתונים.

הפונקציה מחזירה את הטורים: database_id; file_id; sample_ms, שהוא מספר אלפיות השנייה מאז המופע של SQL Server אותחל (ויכול לשמש להשוואת פלטים שונים מפונקציה זו); num_of_reads; num_of_bytes_read; io_stall_read_ms, שהוא סך הזמן, באלפיות שנייה, שהמשתמשים המתינו לקריאות שבוצעו על הקובץ; num_of_writes; num_of_bytes_written; io_stall_write_ms, שהוא משך הזמן, באלפיות שנייה, לו המתינו המשתמשים שה–I/O מול הקובץ יסתיים; size_on_disk_bytes (בבתים); ו–file_handle, שהוא ה– file handle של Microsoft Windows עבור קובץ זה.

שים לב: המדידות מאופסות כאשר SQL Server מאותחל, והן מלמדות רק על I/O פיסי מול הקבצים ולא על I/O לוגי.

בנקודה זו, ברצוננו לגלות אילו מסדי נתונים מערבים I/O ועיכובי I/O הרבים ביותר במערכת, וכן, בתוך מסד הנתונים, אילו סוגי קבצים (נתונים/לוג). השאילתה הבאה תספק לך מידע זה, ממוין בסדר יורד לפי עיכובי I/O, והיא תפיק (על המערכת שלי) את הפלט המוצג בצורה מקוצרת בטבלה 4–3:

```
WITH DBIO AS
(
  SELECT
    DB_NAME(IVFS.database_id) AS db,
    CASE WHEN MF.type = 1 THEN 'log' ELSE 'data' END AS file_type,
    SUM(IVFS.num_of_bytes_read + IVFS.num_of_bytes_written) AS io,
    SUM(IVFS.io_stall) AS io_stall
  FROM sys.dm_io_virtual_file_stats(NULL, NULL) AS IVFS
    JOIN sys.master_files AS MF
      ON IVFS.database_id = MF.database_id
      AND IVFS.file_id = MF.file_id
  GROUP BY DB_NAME(IVFS.database_id), MF.type
)
SELECT db, file_type,
  CAST(1. * io / (1024 * 1024) AS DECIMAL(12, 2)) AS io_mb,
  CAST(io_stall / 1000. AS DECIMAL(12, 2)) AS io_stall_s,
  CAST(100. * io_stall / SUM(io_stall) OVER()
       AS DECIMAL(10, 2)) AS io_stall_pct,
  ROW_NUMBER() OVER(ORDER BY io_stall DESC) AS rn
FROM DBIO
ORDER BY io_stall DESC;
```

טבלה 4–3: מידע I/O של מסד נתונים בצורה מקוצרת

db	*file_type*	*io_mb*	*io_stall_s*	*io_stall_pct*	*rn*
Performance	data	11400.81	11172.55	74.86	1
Performance	log	3732.40	2352.55	15.76	2
tempdb	data	940.08	1327.59	8.90	3
Generic	data	54.10	21.78	0.15	4
master	log	7.33	13.75	0.09	5
tempdb	log	5.24	11.20	0.08	6
master	data	13.56	6.20	0.04	7
Generic	log	1.02	3.81	0.03	8
msdb	data	5.07	2.76	0.02	9
AdventureWorks	log	0.55	1.71	0.01	10
…					

הפלט מציג את שם מסד הנתונים, סוג הקובץ, סך I/O (קריאות וכתיבות) במגה–בתים, עיכובי I/O בשניות, עיכובי I/O כאחוז מסך–הכל במערכת כולה, ומספר שורה המציין מיקום ברשימה הממוינת בהתבסס על עיכובי I/O. כמובן שבאפשרותך, אם רצונך בכך, לחשב אחוז ומספר שורה בהתבסס על I/O בניגוד לעיכובי I/O, וכן באפשרותך להשתמש בשיטות צבירה נעה לחישוב אחוז נע, כפי שהצגתי קודם. ייתכן שתהיה גם מעוניין בהפרדה בין הקריאות והכתיבות לצרכי הניתוח שלך. בפלט זה, תוכל לזהות בבהירות את שלושת האלמנטים העיקריים המעורבים ברוב עיכובי ה–I/O במערכת – חלק הנתונים של Performance, המוביל בגדול; חלק הלוג של Performance; וחלק הנתונים של tempdb. כמובן שעליך להתמקד בשלושה אלמנטים אלו, עם תשומת לב מיוחדת לפעילות נתונים כנגד מסד הנתונים Performance.

בנוגע לעיקר הבעיה שלנו – I/O כנגד חלק הנתונים של מסד הנתונים Performance – כעת עליך להמשיך מטה לרמת העיבוד כדי לזהות את התהליכים המעורבים במרבית ההמתנות.

באשר ל– transaction log, ראשית עליך לבדוק האם הוא מוגדר בצורה הולמת. כלומר, האם הוא ממוקם על דיסק משל עצמו ללא הפרעות, ובמידה שכן, האם הדיסק מהיר מספיק. אם הלוג ממוקם על דיסק איטי, ייתכן שתרצה לשקול הקדשת דיסק מהיר יותר עבורו. ברגע שפעילות הכתיבה ללוג עוברת את ההספק של הדיסק, אתה מתחיל לקבל המתנות ועיכובים. ייתכן שאכן תקדיש דיסק מהיר יותר עבור הלוג, אך שוב, ייתכן שאין לך את התקציב או שיתכן שכבר הקדשת לו את הדיסק המהיר ביותר שבאפשרותך לתת. זכור שה– transaction log נכתב בצורה סדרתית, כך שפריסתו על פני מספר

דיסקים לא תעזור, אלא אם כן יש לך גם פעילויות הקוראות מהלוג (כגון רפליקציה של transaction log, או טריגרים ב– SQL Server 2000). ייתכן גם שתוכל לבצע אופטימיזציה לתהליכים הגורמים לכתיבה אינטנסיבית ללוג על ידי הפחתת כמות הכתיבה שלהן ללוג. ב– transaction log ובאופטימיזציה שלו אדון בהרחבה בפרק 8.

באשר ל–tempdb, קיימות המון פעילויות – הן ישירות והן עקיפות – שעשויות ליצור עומס ב–tempdb עד לנקודה בה הוא עלול להפוך לצוואר בקבוק רציני במערכת. למעשה, פנימית tempdb משמש בצורה כבדה יותר ב– SQL Server 2005 בגלל טכנולוגיית ה– row versioning החדשה המוטמעת במנוע. Row versioning משמש לצורך ה– snapshot isolations החדשים, יצירת טבלאות inserted ו–deleted המיוחדות בטריגרים, פעולות אינדקס online החדשות, והתמיכה החדשה ב– MARS (Multiple Active Result Sets). טכנולוגיית ה– row versioning החדשה מאפשרת תחזוקת מספר גרסאות שורה עקביות, ישנות יותר, ברשימה מקושרת ב–tempdb. לרוב, יש מקום רב לביצוע אופטימיזציה על tempdb, ואין ספק שעליך לתת לאפשרות זו תשומת לב מספקת. על tempdb ועל row versioning ארחיב בספר **תכנות T-SQL** בפרקים המכסים טבלאות זמניות, טריגרים וטרנזקציות.

לצורך ההדגמה שלנו, הבה נתמקד בפתרון בעיות ה–I/O הקשורות לחלק הנתונים של מסד הנתונים Performance.

ממשיכים מטה לרמת התהליך

כעת כאשר אתה יודע אילו מסדי נתונים (במקרה שלנו, אחד) מעורבים במרבית בעיות הביצועים, תרצה להמשיך מטה לרמת התהליך; בפרט, לזהות את התהליכים (פרוצדורות מאוחסנות, שאילתות וכו') אותם יש לכוונן. תמצא שיכולות ה–trace המובנות של SQL Server חזקות ביותר לצורך משימה זו. עליך לעקוב אחר עומס המייצג את הפעילויות הטיפוסיות במערכת מול מסד נתונים עליו עליך להתמקד, לנתח את נתוני ה–trace, ולבודד את התהליכים אותם יש לכוונן.

בטרם אדון ב–trace הספציפי שעליך ליצור למטרות כוונון כאלו, ארצה ראשית לציין מספר טיפים חשובים בקשר לעבודה עם traces ב– SQL Server באופן כללי.

ל–traces יש השפעה על ביצועי המערכת, ועליך להשקיע מאמצים להפחתת ההשפעה שלהם. ידידי הטוב בריאן מוראן השווה פעם את ההיבט הבעייתי של מדידת ביצועים לעיקרון אי–הוודאות של הייסנברג במכניקת הקוונטים. העיקרון נוסח על ידי וורנר הייסנברג (Werner Heisenberg) בשנת 1927. בתרגום חופשי, כאשר אתה מודד משהו, קיים גורם אי–ודאות הנגרם על ידי המדידה שלך. ככל שהמדידה של מיקומו של דבר יותר מדויקת, כך גדלה אי הוודאות הקיימת בנוגע למומנטום שלו (בצורה חופשית, מהירות וכיוון). כך שככל שאתה יודע דבר אחד בצורה מדויקת, כך יגדל חוסר הדיוק שלך לגבי ישות מקבילה כלשהי. בקנה מידה של אטומים וחלקיקים בסיסיים, ההשפעה

של עיקרון אי–הוודאות חשובה ביותר. לא קיימת הוכחה לתמיכה בעיקרון אי–הוודאות (ממש כשם שאין הוכחה מדעית לאלוהים או לאבולוציה), אך התיאוריה מבוססת מתמטית ונתמכת על ידי ניסויים.

נחזור ל–traces שלנו, אינך מעוניין שפעילות ה–trace שלך תגרום לבעיית ביצועים בפני עצמה. אינך יכול להימנע מההשפעה שלה לחלוטין – דבר זה בלתי אפשרי – אך אתה בהחלט יכול לעשות הרבה להפחית אותה על ידי שמירת מספר קווים מנחים חשובים:

- אל תבצע trace עם ה–GUI של Profiler; במקום זאת, השתמש בקוד ה–T-SQL המגדיר את ה–trace. כאשר אתה מבצע trace עם Profiler, אתה מריץ למעשה שני traces – אחד המכוון את הפלט לקובץ היעד, ואחד המזרים את מידע ה–trace ללקוח המריץ את Profiler. ניתן להגדיר את ה–trace גרפית עם Profiler, ואז לייצר מהגדרת ה–trace קוד T-SQL על ידי שימוש באפשרות התפריט File>Export>Script Trace Definition>For SQL Server 2005... (or 2000...). תוכל לבצע שינויים קלים לקוד בהתאם לצרכיך. אני אוהב לעטוף את הקוד בפרוצדורה מאוחסנת, המקבלת כארגומנטים אלמנטים בהם אני מעוניין כמשתנים – למשל, קוד מסד הנתונים בו השתמשתי כמסנן בהגדרת ה–trace.
- אל תבצע trace ישירות לטבלה, שכן תהיה לכך השפעה משמעותית על הביצועים. ביצוע trace לקובץ על דיסק מקומי היא האפשרות המהירה ביותר (ביצוע trace לקובץ יעד הנמצא בשיתוף ברשת הוא גרוע גם כן). תוכל אחר–כך לטעון את מידע ה–trace לטבלה לצורך ניתוח על ידי שימוש בפונקציה fn_trace_gettable עם משפט SELECT INTO. SELECT INTO נחשב לפעולת bulk. פעולות bulk נרשמות בלוג בצורה מינימלית כאשר מודל ההתאוששות (recovery model) של מסד הנתונים אינו מוגדר כ–FULL; לפיכך, הן רצות הרבה יותר מהר מכפי שהן רצות במצב של רישום בלוג בצורה מלאה.
- ביצוע trace עשוי לייצר כמות אדירה של נתונים ופעילות I/O מוגברת. ודא שקובץ היעד של ה–trace לא ממוקם על דיסקים המכילים קבצי מסד נתונים (כגון נתונים, לוג ו–tempdb). אידיאלית, רצוי להקדיש דיסק נפרד עבור קבצי היעד של ה–trace, גם אם המשמעות היא הוספת דיסקים חיצוניים.
- היה סלקטיבי בבחירת מחלקות אירועים וטורי נתונים – בצע trace רק על מה שצריך, הסר את כל ברירות המחדל ואת כל האירועים והטורים הבלתי–חיוניים. כמובן אל תהיה סלקטיבי מדי; ודא שכל מחלקות האירועים וטורי הנתונים הרלוונטיים כלולים.
- השתמש ביכולות הסינון של ה–trace לסינון האירועים הרלוונטיים בלבד. למשל, כאשר אתה מכוונן מסד נתונים מסוים, ודא שאתה מסנן אירועים רק עבור קודי מסדי הנתונים הרלוונטיים.

בעודנו זוכרים את הקווים המנחים הללו, הבה נמשיך ל–trace לו אנו זקוקים למטרות הכוונון שלנו.

Trace ביצועי פעילות

כעת עליך להגדיר trace שיסייע לך לזהות תהליכים שיש לכוונן במסד הנתונים Performance. כאשר יש בפניך צורך כזה, קיימת נטייה לבצע trace על תהליכים שרצים לאט על ידי סינון אירועים בהם טור הנתונים Duration גדול או שווה לערך מסוים (נאמר, 3000 אלפיות שנייה). גישה זו בעייתית. חשוב על המקרה הבא: יש לך שאילתה שרצה כ–30 שניות מספר פעמים ביום, ושאילתה אחרת שרצה כחצי שנייה 40,000 פעמים ביום. אילו מהן היית אומר שחשוב יותר לכוונן? כמובן שהאחרונה יותר חשובה, אך אם אתה מסנן רק אירועים הרצים לפחות שלוש שניות, תסנן החוצה את השאילתה החשובה יותר לכוונון.

בקיצור, למטרות שלנו כלל אינך רוצה לסנן בהתבסס על Duration. כמובן, המשמעות היא שאתה עשוי לקבל כמות עצומה של נתוני trace, כך שרצוי שתקפיד על הקווים המנחים שהצעתי קודם. אתה כן רוצה לסנן רק מסדי נתונים הרלוונטיים לתהליך הכוונון שלך.

באשר למחלקות אירועים, אם מרבית הפעילויות במערכת שלך נקראות על ידי פרוצדורות מאוחסנות וכל פרוצדורה מאוחסנת קוראת למספר קטן או מוגבל של פעילויות, בצע trace על מחלקת האירועים SP:Completed. אז תוכל לצבור את הנתונים לפי פרוצדורה. לעומת זאת, אם כל פרוצדורה קוראת לפעילויות רבות, תרצה לבצע trace על מחלקת האירועים SP:StmtCompleted כדי ללכוד כל משפט יחידי הנקרא מכל פרוצדורה מאוחסנת. אם יש לך פעילויות הנשלחות כאד הוק batches (כמו במקרה שלנו), בצע trace על מחלקת האירועים SQL:StmtCompleted. לבסוף, אם יש לך פעילויות הנשלחות כקריאות פרוצדורה מרוחקות (remote procedure calls), בצע מעקב על מחלקת האירועים RPC:Completed. שים לב שכל מחלקות האירועים הן Completed בניגוד למחלקות האירועים Starting התואמות. רק אירועי מחלקה Completed מחזיקים מידע ביצועים כמו Duration, CPU, Reads, Writes משום שטבעית, ערכים אלו אינם ידועים כאשר האירועים התואמים מתחילים.

באשר לטורי נתונים, אתה צריך בעיקר את הטור TextData שיחזיק את קוד ה–T-SQL עצמו, ואת מוני הביצועים הרלוונטיים – החשוב ביותר, הטור Duration. זכור שמשתמשים תופסים את ההמתנות כבעיית הביצועים, ו–Duration מייצג את הזמן שלוקח לאירוע לרוץ. אני אוהב גם לבצע trace על טור הנתונים RowCounts, במיוחד כאשר אני מחפש בעיות הקשורות ברשת. שאילתות המחזירות את התוצאה ללקוח עם מספרים גדולים במונה זה יעידו על עומס אפשרי על הרשת. מעבר לכך, ייתכן שתרצה טורי נתונים נוספים בהתבסס על צרכיך. למשל, אם תרצה בשלב מאוחר יותר לנתח את הנתונים לפי עמדת לקוח (host), יישום, login וכו', ודא שאתה כולל גם את טורי הנתונים המתאימים.

ניתן להגדיר trace בהתאם לקווים מנחים אלו, ואז לייצא את הגדרת ה–trace לקוד T-SQL. זה מה שאני עשיתי, ועטפתי את הקוד בפרוצדורה מאוחסנת הנקראת sp_perfworkload_trace_start.

שים לב: מיקרוסופט לא ממליצה להשתמש בתחילית _sp עבור פרוצדורות מאוחסנות מקומיות של משתמש. יצרתי את הפרוצדורה המאוחסנת עם התחילית _sp במסד הנתונים master משום שזה הופך את הפרוצדורה ל"מיוחדת", במובן שאתה יכול לקרוא לה מכל מסד נתונים מבלי לציין את שם מסד הנתונים כתחילית.

הפרוצדורה המאוחסנת מקבלת את קוד מסד הנתונים ואת שם הקובץ כפרמטרי קלט. היא מגדירה trace על ידי שימוש בקוד מסד הנתונים המסופק כמסנן, ובשם הקובץ הנתון כיעד עבור נתוני ה–trace; היא מתחילה את ה–trace, ומחזירה את קוד ה–trace שזה עתה נוצר דרך פרמטר פלט. הרץ את הקוד בקטע–קוד 3–3 ליצירת הפרוצדורה המאוחסנת sp_perfworkload_trace_start.

קטע–קוד 3–3: קוד יצירה לפרוצדורה המאוחסנת **sp_perfworkload_trace_start**

```
SET NOCOUNT ON;
USE master;
GO

IF OBJECT_ID('dbo.sp_perfworkload_trace_start') IS NOT NULL
  DROP PROC dbo.sp_perfworkload_trace_start;
GO

CREATE PROC dbo.sp_perfworkload_trace_start
  @dbid     AS INT,
  @tracefile AS NVARCHAR(254),
  @traceid  AS INT OUTPUT
AS

-- Create a Queue
DECLARE @rc         AS INT;
DECLARE @maxfilesize AS BIGINT;

SET @maxfilesize = 5;

EXEC @rc = sp_trace_create @traceid OUTPUT, 0, @tracefile, @maxfilesize, NULL
IF (@rc != 0) GOTO error;

-- Client side File and Table cannot be scripted

-- Set the events
DECLARE @on AS BIT;
```

```
SET @on = 1;
EXEC sp_trace_setevent @traceid, 10, 15, @on;
EXEC sp_trace_setevent @traceid, 10, 8, @on;
EXEC sp_trace_setevent @traceid, 10, 16, @on;
EXEC sp_trace_setevent @traceid, 10, 48, @on;
EXEC sp_trace_setevent @traceid, 10, 1, @on;
EXEC sp_trace_setevent @traceid, 10, 17, @on;
EXEC sp_trace_setevent @traceid, 10, 10, @on;
EXEC sp_trace_setevent @traceid, 10, 18, @on;
EXEC sp_trace_setevent @traceid, 10, 11, @on;
EXEC sp_trace_setevent @traceid, 10, 12, @on;
EXEC sp_trace_setevent @traceid, 10, 13, @on;
EXEC sp_trace_setevent @traceid, 10, 14, @on;
EXEC sp_trace_setevent @traceid, 45, 8, @on;
EXEC sp_trace_setevent @traceid, 45, 16, @on;
EXEC sp_trace_setevent @traceid, 45, 48, @on;
EXEC sp_trace_setevent @traceid, 45, 1, @on;
EXEC sp_trace_setevent @traceid, 45, 17, @on;
EXEC sp_trace_setevent @traceid, 45, 10, @on;
EXEC sp_trace_setevent @traceid, 45, 18, @on;
EXEC sp_trace_setevent @traceid, 45, 11, @on;
EXEC sp_trace_setevent @traceid, 45, 12, @on;
EXEC sp_trace_setevent @traceid, 45, 13, @on;
EXEC sp_trace_setevent @traceid, 45, 14, @on;
EXEC sp_trace_setevent @traceid, 45, 15, @on;
EXEC sp_trace_setevent @traceid, 41, 15, @on;
EXEC sp_trace_setevent @traceid, 41, 8, @on;
EXEC sp_trace_setevent @traceid, 41, 16, @on;
EXEC sp_trace_setevent @traceid, 41, 48, @on;
EXEC sp_trace_setevent @traceid, 41, 1, @on;
EXEC sp_trace_setevent @traceid, 41, 17, @on;
EXEC sp_trace_setevent @traceid, 41, 10, @on;
EXEC sp_trace_setevent @traceid, 41, 18, @on;
EXEC sp_trace_setevent @traceid, 41, 11, @on;
EXEC sp_trace_setevent @traceid, 41, 12, @on;
EXEC sp_trace_setevent @traceid, 41, 13, @on;
EXEC sp_trace_setevent @traceid, 41, 14, @on;

-- Set the Filters
DECLARE @intfilter AS INT;
DECLARE @bigintfilter AS BIGINT;
-- Application name filter
```

```
EXEC sp_trace_setfilter @traceid, 10, 0, 7, N'SQL Server Profiler%';
-- Database ID filter
EXEC sp_trace_setfilter @traceid, 3, 0, 0, @dbid;

-- Set the trace status to start
EXEC sp_trace_setstatus @traceid, 1;

-- Print trace id and file name for future references
PRINT 'Trce ID: ' + CAST(@traceid AS VARCHAR(10))
  + ', Trace File: ''' + @tracefile + '''';

GOTO finish;

error:
PRINT 'Error Code: ' + CAST(@rc AS VARCHAR(10));

finish:
GO
```

הרץ את הקוד הבא כדי להתחיל את ה–trace, לסנן אירועים ממסד הנתונים Performance ולשלוח את נתוני ה–trace לקובץ 'c:\temp\Perfworkload 20060828.trc':

```
DECLARE @dbid AS INT, @traceid AS INT;
SET @dbid = DB_ID('Performance');

EXEC dbo.sp_perfworkload_trace_start
  @dbid     = @dbid,
  @tracefile = 'c:\temp\Perfworkload 20060828.trc',
  @traceid  = @traceid OUTPUT;
```

אם נניח שקוד ה–trace שזה עתה נוצר הוא 2, תקבל את הפלט הבא:

```
Trace ID: 2, Trace File: 'c:\temp\perfworkload 20060828.trc'
```

עליך לשמור את קוד ה–trace בצד, שכן תשתמש בו מאוחר יותר כדי לעצור את ה–trace ולסגור אותו.

כעת, הרץ את שאילתות הדוגמה מקטע–קוד 2–3 מספר פעמים. כשתסיים, עצור את ה–trace וסגור אותו על ידי הרצת הקוד הבא (בהנחה שקוד ה–trace הוא 2):

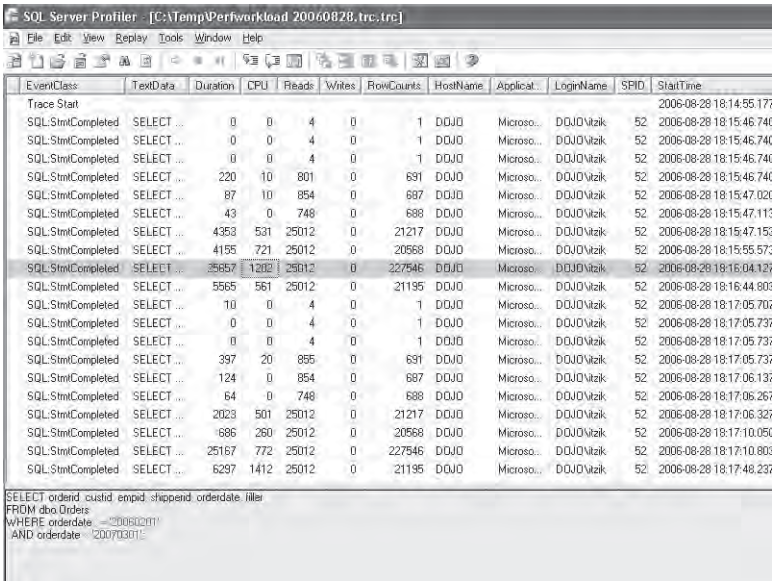
```
EXEC sp_trace_setstatus 2, 0;
EXEC sp_trace_setstatus 2, 2;
```

כמובן, עליך לציין את קוד ה–trace שקיבלת בפועל עבור ה–trace שלך. אם איבדת את הפתק עליו כתבת את קוד ה–trace, בצע שאילתה על ה– sys.traces view לקבלת מידע על כל ה–traces שרצים.

כאשר אתה מבצע trace על עומס במערכת תפעולית למטרות כוונון, ודא שאתה מבצע trace על אחד שהוא מייצג מספיק. במקרים מסוימים, המשמעות היא ביצוע trace רק למספר שעות, בעוד שבמקרים אחרים הוא עשוי להימשך ימים.

הצעד הבא הוא לטעון את נתוני ה–trace לטבלה ולנתח אותה. כמובן, אתה יכול לפתוח אותם עם Profiler ולבחון אותם שם; אך לרוב, traces כאלו מייצרים נתונים רבים ואינך יכול לעשות הרבה עם Profiler לצרכי ניתוח. במקרה שלנו, יש לנו מספר קטן של שאילתות דוגמה. תרשים 4–3 מציג כיצד נראים נתוני ה–trace כאשר מעלים אותם ל–Profiler.

תרשים 4–3: נתוני trace של ביצועי פעילות



בחן את נתוני ה–trace, ניתן לראות בבירור כמה שאילתות בעלות זמן ריצה ממושך המייצרות כמות גדולה של I/O. שאילתות אלו משתמשות במסנני תחום בהתבסס על הטור orderdate ונראה שהן גורמות באופן קבוע לכ–25,000 קריאות. טבלת Orders כרגע מכילה 1,000,000 שורות ומשתרעת על פני כ–25,000 דפים. דבר זה אומר לך ששאילתות אלו גורמות לסריקות מלאות של הטבלה למציאת הנתונים וקרוב לוודאי שחסר להן אינדקס חשוב על הטור orderdate. סביר להניח שהאינדקס החסר הוא הסיבה העיקרית לכמות ה–I/O המופרזת במערכת.

כמו כן, ניתן למצוא שאילתות המחזירות מספר שורות גדול מאוד בתוצאה–מספר אלפים, ובמקרים מסוימים, מאות אלפי שורות. ככל שניתן, עדיף לבדוק האם מסננים ומניפולציות נוספות מופעלים בשרת, מאשר להביא הכל ללקוח דרך הרשת ולבצע סינון ומניפולציות נוספות שם. שאילתות אלו הן אולי הסיבה העיקרית לבעיות הרשת במערכת.

כמובן, ניתוח גרפי כזה עם Profiler ישים רק עם traces קטנים מאוד, כמו זה בו אנו משתמשים למטרות הדגמה. במערכות תפעוליות, הדבר פשוט לא מציאותי; תצטרך להעלות את נתוני ה–trace לטבלה ולהשתמש בשאילתות לניתוח הנתונים.

ניתוח נתוני Trace

כפי שהזכרתי קודם, אתה משתמש בפונקציה fn_trace_gettable כדי להחזיר את נתוני ה–trace במבנה טבלה. הרץ את הקוד הבא כדי לטעון את נתוני ה–trace מהקובץ שלנו לטבלת Workload:

```
SET NOCOUNT ON;
USE Performance;
GO
IF OBJECT_ID('dbo.Workload') IS NOT NULL
  DROP TABLE dbo.Workload;
GO

SELECT CAST(TextData AS NVARCHAR(MAX)) AS tsql_code,
  Duration AS duration
INTO dbo.Workload
FROM sys.fn_trace_gettable('c:\temp\Perfworkload 20060828.trc', NULL) AS T
WHERE Duration IS NOT NULL;
```

שים לב שקוד זה טוען רק את הנתונים מהטורים TextData (קוד T-SQL) ו–Duration כדי להתמקד במיוחד בזמן הריצה של השאילתה. לרוב, תרצה להעלות גם טורי נתונים אחרים הרלוונטיים לניתוח שלך – למשל, מוני ה–I/O והמעבד, ספירת שורות, שם host, שם יישום, וכו'.

זכור שחשוב לצבור את מידע הביצועים על ידי השאילתה או משפט ה–T-SQL כדי למצוא את ההשפעה הכללית על הביצועים של כל שאילתה עם ההפעלות המרובות שלה. הקוד הבא מנסה לעשות זאת ומייצר את הפלט המוצג בצורה מקוצרת בטבלה 5–3:

```
SELECT
  tsql_code,
  SUM(duration) AS total_duration
FROM dbo.Workload
GROUP BY tsql_code;
```

טבלה 5–3: צבירה של Duration לשאילתה בצורה מקוצרת

tsql_code	*duration*

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '20060118';`
	161055

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '20060212';`
	367430

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '20060828';`
	132466

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '20060101'`
`  AND orderdate < '20060201';`
	3821240

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '20060201'`
`  AND orderdate < '20070301';`
	47881415

...	

אך קיימת בעיה. ניתן לראות בנתוני הצבירה שכמה שאילתות הזהות לוגית או שיש להן תבנית זהה מופיעות בקבוצות שונות. זאת משום שהן משתמשות בערכים שונים במסננים שלהם. רק מחרוזות שאילתה זהות לחלוטין קובצו יחד. כהערת ביניים, בבעיה זו לא היית נתקל אם היית משתמש בפרוצדורות מאוחסנות, שכל אחת קוראת לשאילתה נפרדת או למספר קטן מאוד של שאילתות. זכור שבמקרה כזה היית מבצע trace על מחלקת האירועים SP:Completed, ואז היית מקבל נתונים מצטברים לפרוצדורה. אך זהו לא המקרה שלנו.

דרך פשוטה אך לא מאוד מדויקת להתמודד עם הבעיה, היא לחלץ תת–מחרוזת של מחרוזות השאילתה ולצבור נתונים לתת–מחרוזת זו. לרוב, החלק השמאלי של מחרוזות שאילתה בעלות אותה תבנית הוא זהה, בעוד שבמקום כלשהו מימין מופיעים הארגומנטים המשמשים לסינון. תוכל להפעיל שיטת ניסוי וטעייה, תוך שאתה משחק עם אורך תת–המחרוזת שתחלץ; בשאיפה שתת–המחרוזת תהיה ארוכה מספיק כדי לאפשר קיבוץ שאילתות בעלות תבנית זהה יחד, וקצרה מספיק כדי להבחין בין שאילתות בעלות תבנית שונה. גישה זו, כפי שניתן לראות, נפתלת ואינה מבטיחה תוצאות מדויקות. למעשה, אתה בוחר מספר שנראה סביר, עוצם את עיניך ומקווה לטוב.

למשל, השאילתה הבאה צוברת את נתוני ה–trace על ידי תחילית שאילתה של 100 תווים ומפיקה את הפלט המוצג בטבלה 6–3:

```
SELECT
  SUBSTRING(tsql_code, 1, 100) AS tsql_code,
  SUM(duration) AS total_duration
FROM dbo.Workload
GROUP BY SUBSTRING(tsql_code, 1, 100);
```

טבלה 6–3: Duration מצטבר לתחילת שאילתה

tsql_code	*total_duration*
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '200	660951
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20	60723155
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid = 3;	17857
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid = 5;	426
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid = 7;	598
SET NOCOUNT ON;	7
USE Performance;	12857

במקרה שלנו, אורך התחילית השיגה את התוצאה הרצויה עבור שאילתות מסוימות, אך לא עבור אחרות. עם נתוני trace מציאותיים יותר, לא תהיה לך את הפריבילגיה להתבונן במספר קטנטן של שאילתות ולשחק עם המספרים בכזו קלות. אך הרעיון הכללי הוא שאתה מכוונן את אורך התחילית על ידי ניסוי וטעייה. להלן הקוד המשתמש באורך תחילית 94 ומפיק את הפלט המוצג בטבלה 7–3:

```
SELECT
  SUBSTRING(tsql_code, 1, 94) AS tsql_code,
  SUM(duration) AS total_duration
FROM dbo.Workload
GROUP BY SUBSTRING(tsql_code, 1, 94);
```

טבלה 3–7: Duration מצטבר לתחילית שאילתה, לאחר כוונון אורך התחילית

tsql_code	*total_duration*
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate	61384106
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid =	18881
SET NOCOUNT ON;	7
USE Performance;	12857

כעת יש לך קיבוץ יתר. בקצרה, מציאת אורך התחילית הנכון הוא תהליך נפתל, ורמת הדיוק והביטחון מוטלת בספק.

גישה מדויקת הרבה יותר היא לפרק את מחרוזות השאילתה ולייצר חתימת שאילתה (query signature) לכל אחת. חתימת שאילתה היא תבנית שאילתה הזהה לשאילתות בעלות תבנית זהה. לאחר שיצרת את אלו, תוכל אז לצבור את הנתונים לחתימות שאילתה במקום למחרוזות השאילתה עצמן. SQL Server 2005 מספק לך את הפרוצדורה המאוחסנת sp_get_query_template, המפרקת מחרוזת שאילתת קלט ומחזירה את תבנית השאילתה ואת ההגדרה של הארגומנטים דרך פרמטרי פלט.

כדוגמה, הקוד הבא קורא לפרוצדורה המאוחסנת, מספק מחרוזת שאילתה לדוגמה כקלט, ומפיק את הפלט המוצג בטבלה 3–8:

```
DECLARE @my_templatetext AS NVARCHAR(MAX);
DECLARE @my_parameters   AS NVARCHAR(MAX);

EXEC sp_get_query_template
  N'SELECT * FROM dbo.T1 WHERE col1 = 3 AND col2 > 78',
  @my_templatetext OUTPUT,
  @my_parameters OUTPUT;

SELECT @my_templatetext AS querysig, @my_parameters AS params;
```

טבלה 3–8: תבנית שאילתה

querysig	*params*
select * from dbo.T1 where col1 = @0 and col2 > @1	@0 int,@1 int

הבעיה עם פרוצדורה מאוחסנת זו היא שאתה צריך להשתמש בסמן כדי לקרוא לה מול כל מחרוזת שאילתה מנתוני ה–trace, ודבר זה עשוי לארוך זמן רב למדי ב–traces גדולים. הפרוצדורה המאוחסנת עוצבה גם להחזיר שגיאה במקרים מסוימים (ראה Books Online לפרטים), דבר שעלול להפחית מערכה. יהיה נוח הרבה יותר ליישם לוגיקה זו כפונקציה, המאפשרת לך לקרוא לה ישירות מול הטבלה המכילה את נתוני ה–trace. למזלנו, פונקציה כזו קיימת; היא נכתבה על ידי סטיוארט עוזר (Stuart Ozer), והקוד שלה מופיע בקטע–קוד 4–3. סטיוארט עובד בצוות הייעוץ ללקוחות של Microsoft SQL Server, ואני רוצה להודות לו על שהרשה לי לחלוק את הקוד עם קוראי הספר.

קטע–קוד 4–3: קוד יצירה של הפונקציה fn_SQLSigTSQL

```
IF OBJECT_ID('dbo.fn_SQLSigTSQL') IS NOT NULL
  DROP FUNCTION dbo.fn_SQLSigTSQL;
GO

CREATE FUNCTION dbo.fn_SQLSigTSQL
  (@p1 NTEXT, @parselength INT = 4000)
RETURNS NVARCHAR(4000)

--
-- This function is provided "AS IS" with no warranties,
-- and confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- Strips query strings
AS
BEGIN
  DECLARE @pos AS INT;
  DECLARE @mode AS CHAR(10);
  DECLARE @maxlength AS INT;
  DECLARE @p2 AS NCHAR(4000);
  DECLARE @currchar AS CHAR(1), @nextchar AS CHAR(1);
  DECLARE @p2len AS INT;

  SET @maxlength = LEN(RTRIM(SUBSTRING(@p1,1,4000)));
  SET @maxlength = CASE WHEN @maxlength > @parselength
                     THEN @parselength ELSE @maxlength END;
  SET @pos = 1;
  SET @p2 = '';
  SET @p2len = 0;
  SET @currchar = '';
```

```
  set @nextchar = '';
  SET @mode = 'command';

  WHILE (@pos <= @maxlength)
  BEGIN
    SET @currchar = SUBSTRING(@p1,@pos,1);
    SET @nextchar = SUBSTRING(@p1,@pos+1,1);
    IF @mode = 'command'
    BEGIN
      SET @p2 = LEFT(@p2,@p2len) + @currchar;
      SET @p2len = @p2len + 1 ;
      IF @currchar IN (',','(',' ','=','<','>','!')
        AND @nextchar BETWEEN '0' AND '9'
      BEGIN
        SET @mode = 'number';
        SET @p2 = LEFT(@p2,@p2len) + '#';
        SET @p2len = @p2len + 1;
      END
      IF @currchar = ''''
      BEGIN
        SET @mode = 'literal';
        SET @p2 = LEFT(@p2,@p2len) + '#''';
        SET @p2len = @p2len + 2;
      END
    END
    ELSE IF @mode = 'number' AND @nextchar IN (',',')',' ','=','<','>','!')
      SET @mode= 'command';
    ELSE IF @mode = 'literal' AND @currchar = ''''
      SET @mode= 'command';

    SET @pos = @pos + 1;
  END
  RETURN @p2;
END
GO
```

הפונקציה מקבלת כקלטים מחרוזת שאילתה ואת אורך הקוד שברצונך לפרק. הפונקציה מחזירה את חתימת השאילתה של שאילתת הקלט, כאשר כל הפרמטרים מוחלפים בתו סולמית (#). שים לב שזוהי פונקציה פשוטה למדי ויתכן שתצטרך לעבור התאמות עבור סיטואציות מסוימות. הרץ את הקוד הבא כדי לנסות את הפונקציה:

```
SELECT dbo.fn_SQLSigTSQL
  (N'SELECT * FROM dbo.T1 WHERE col1 = 3 AND col2 > 78', 4000);
```

תקבל את הפלט הבא:

```
SELECT * FROM dbo.T1 WHERE col1 = # AND col2 > #
```

כמובן, כעת באפשרותך להשתמש בפונקציה ולצבור את נתוני ה–trace לפי חתימת שאילתה. עם זאת, זכור שעל אף ש–T-SQL יעיל מאוד במניפולציית נתונים, הוא איטי בעיבוד לוגיקה איטרטיבית/פרוצדורלית. זוהי דוגמה קלאסית בה יישום CLR של הפונקציה מתאים יותר. ה–CLR מהיר הרבה יותר מאשר T-SQL עבור לוגיקה איטרטיבית/פרוצדורלית ומניפולציית מחרוזות. SQL Server 2005 מציג שילוב NET. מובנה במוצר, המאפשר לך לפתח רוטינות NET. בהתבסס על CLR (Common Language Runtime). רוטינות CLR נדונות בספר Inside T-SQL Programming, שם תמצא כיסוי מלא יותר והשוואה בין יישום ה–T-SQL והיישום מבוסס ה–CLR של הפונקציה המייצרת חתימות שאילתה. תוכל גם למצוא מידע רקע על פיתוח CLR ב– SQL Server בפרק 6 של ספר זה.

היישום ה"מורחב" מבוסס ה–CLR של הפונקציה המשתמש בקוד #C מוצג בקטע–קוד 5–3.

קטע–קוד 5–3: הפונקציות fn_SQLSigCLR ו–fn_RegexReplace, גרסת #C

```
using System.Text;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;
using System.Text.RegularExpressions;

public partial class SQLSignature
{
   // fn_SQLSigCLR
   [SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
   public static SqlString fn_SQLSigCLR(SqlString querystring)
   {
      return (SqlString)Regex.Replace(
         querystring.Value,
         @"([\s,(=<>!](?![^\]]+[\]]))(?:(?:(?:(?#    expression coming
          )(?:([N])?(')(?:[^']|'')*('))(?#           character
          )|(?:0x[\da-fA-F]*)(?#                     binary
          )|(?:[-+]?(?:(?:[\d]*\.[\d]*|[\d]+)(?#     precise number
          )(?:[eE]?[\d]*)))(?#                       imprecise number
          )|(?:[~]?[-+]?(?:[\d]+))(?#                integer
          ))(?:[\s]?[\+\-\*\/\%\&\|\^][\s]?)?)+(?#   operators
          ))",
         @"$1$2$3#$4");
   }
```

```
    // fn_RegexReplace - for generic use of RegEx-based replace
    [SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
    public static SqlString fn_RegexReplace(
        SqlString input, SqlString pattern, SqlString replacement)
    {
        return (SqlString)Regex.Replace(
            input.Value, pattern.Value, replacement.Value);
    }
}
```

הקוד בקטע–קוד 5–3 מכיל הגדרות של שתי פונקציות: fn_SQLSigCLR ו– fn_RegexReplace. הפונקציה fn_SQLSigCLR מקבלת מחרוזת שאילתה ומחזירה את חתימת השאילתה. הפונקציה מכסה מקרים מהם התעלמה פונקציית ה–T-SQL, וניתן לשפר אותה בקלות לתמיכה במקרים נוספים אם יש לך צורך בכך. הפונקציה fn_RegexReplace חושפת יכולות החלפת מחרוזות מבוססות–תבנית כלליות יותר בהתבסס על regular expressions, ומסופקת כאן לנוחיותך.

שים לב: לא טרחתי לבדוק קלטי NULL בקוד ה–CLR משום ש–T-SQL מאפשר לך להגדיר את האפשרות RETURNS NULL ON NULL INPUT כאשר אתה רושם את הפונקציות, כפי שאדגים בהמשך. המשמעות של אפשרות זו היא שכאשר מסופק קלט NULL, SQL Server כלל לא קורא לפונקציה; אלא הוא פשוט מחזיר פלט NULL.

אם אתה מכיר את הנושא של פיתוח רוטינות CLR ב– SQL Server, הטמע פונקציות אלו במסד הנתונים Performance. אם לא, פשוט עקוב אחר הצעדים הבאים:

1. צור פרויקט Microsoft Visual C#, Class Library חדש ב– Microsoft Visual Studio 2005 (File>New>Project…>Visual C#>Class Library).
2. בתיבת הדו–שיח New Project, קרא לפרויקט ולפתרון **SQLSignature**, ציין C:\ עבור המיקום, ואשר.
3. שנה את שם הקובץ Class1.cs ל–**SQLSignature.cs**, והדבק בתוכו את הקוד מקטע–קוד 5–3, במקום התוכן הקיים.
4. בנה את ה–assembly על ידי בחירה בפריט התפריט Build>Build SQLSignature. קובץ בשם C:\SQLSignature\SQLSignature\bin\Debug\SQLSignature.dll המכיל את ה–assembly ייווצר.

5. בנקודה זו, אתה חוזר ל– SSMS (SQL Server Management Studio) ומפעיל שני צעדים נוספים לצורך הטמעת ה–assembly במסד הנתונים Performance, ורישום של הפונקציות fn_SQLSigCLR ו–fn_RegexReplace. אך ראשית, עליך לאפשר CLR ב– SQL Server (הנמצא במצב disabled כברירת מחדל) על ידי הרצת הקוד הבא:

```
EXEC sp_configure 'clr enable', 1;
RECONFIGURE;
```

6. כעת, עליך לטעון את קוד ה– IL (Intermediate Language) מקובץ ה–dll. לתוך מסד הנתונים Performance על ידי הרצת הקוד הבא:

```
USE Performance;
CREATE ASSEMBLY SQLSignature
FROM 'C:\SQLSignature\SQLSignature\bin\Debug\SQLSignature.dll';
```

7. לבסוף, רשום את הפונקציות fn_SQLSigCLR ו–fn_RegexReplace על ידי הרצת הקוד הבא:

```
CREATE FUNCTION dbo.fn_SQLSigCLR(@querystring AS NVARCHAR(MAX))
RETURNS NVARCHAR(MAX)
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME SQLSignature.SQLSignature.fn_SQLSigCLR;
GO

CREATE FUNCTION dbo.fn_RegexReplace(
  @input       AS NVARCHAR(MAX),
  @pattern     AS NVARCHAR(MAX),
  @replacement AS NVARCHAR(MAX))
RETURNS NVARCHAR(MAX)
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME SQLSignature.SQLSignature.fn_RegexReplace;
GO
```

סיימת. בנקודה זו, באפשרותך להתחיל להשתמש בפונקציות כפי שאתה משתמש בכל פונקציה מוגדרת–משתמש אחרת. אם אתה סקרן לדעת, יישום ה–CLR של הפונקציה רץ פי 10 מהר יותר מאשר זה של T-SQL.

כדי לבדוק את הפונקציה fn_SQLSigCLR, קרא לה מול הטבלה Workload על ידי הרצת השאילתה הבאה, אשר תפיק את הפלט המוצג בצורה מקוצרת בטבלה 9–3:

```
SELECT
  dbo.fn_SQLSigCLR(tsql_code) AS sig,
  duration
FROM dbo.Workload;
```

תוכל גם להשתמש בפונקציה הכללית יותר fn_RegexReplace כדי לקבל פלט זהה כך:

```
SELECT
  dbo.fn_RegexReplace(tsql_code,
    N'([\s,(=<>!](?![^\]]+[\]]))(?:(?:(?:(?#      expression coming
     )(?:([N])?('')(?:[^'']|'''')*(''))(?#        character
     )|(?:0x[\da-fA-F]*)(?#                       binary
     )|(?:[-+]?(?:(?:[\d]*\.[\d]*|[\d]+)(?#       precise number
     )(?:[eE]?[\d]*)))(?#                         imprecise number
     )|(?:[~]?[-+]?(?:[\d]+))(?#                  integer
     ))(?:[\s]?[\+\-\*\/\%\&\|\^][\s]?)?)+(?#     operators
     ))',
    N'$1$2$3#$4') AS sig,
  duration
FROM dbo.Workload;
```

טבלה 9–3: נתוני Trace לחתימות שאילתה בצורה מקוצרת

sig	*duration*
...	
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid = #;	17567
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid = #;	72
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid = #;	145
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '#';	99204
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '#';	56191
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '#';	38718

sig	*duration*

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '#'`
`  AND orderdate < '#';`
	1689740

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '#'`
`  AND orderdate < '#';`
	738292

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '#'`
`  AND orderdate < '#';`
	13601583

...	

כפי שתוכל לראות, אתה מקבל חזרה חתימות שאילתה, בהן אתה יכול להשתמש לצבירת נתוני ה–trace. עם זאת, זכור שמחרוזות שאילתה עלולות להיות ארוכות, וקיבוץ הנתונים לפי מחרוזות ארוכות הוא איטי ויקר. במקום זאת, ייתכן שתעדיף לייצר checksum מספרי לכל מחרוזת שאילתה על ידי שימוש בפונקציית ה–T-SQL – CHECKSUM. למשל, השאילתה הבאה מייצרת ערך checksum לכל מחרוזת שאילתה מטבלת Workload, והיא מייצרת את הפלט המוצג בצורה מקוצרת בטבלה 10–3:

```
SELECT
  CHECKSUM(dbo.fn_SQLSigCLR(tsql_code)) AS cs,
  duration
FROM dbo.Workload;
```

טבלה 10–3: Checksums של חתימות שאילתה בצורה מקוצרת

cs	*duration*
...	
-1872968693	17567
-1872968693	72
-1872968693	145
-184235228	99204
-184235228	56191
-184235228	38718

cs	*duration*
368623506	1689740
368623506	738292
368623506	13601583
...	

השתמש בקוד הבא להוספת טור הנקרא cs לטבלת Workload, מילויו ב–checksum של חתימות השאילתה ויצירת אינדקס–clustered על הטור cs:

```
ALTER TABLE dbo.Workload ADD cs INT NOT NULL DEFAULT (0);
GO
UPDATE dbo.Workload
  SET cs = CHECKSUM(dbo.fn_SQLSigCLR(tsql_code));

CREATE CLUSTERED INDEX idx_cl_cs ON dbo.Workload(cs);
```

הרץ את הקוד הבא להחזרת התוכן החדש של טבלת Workload, המוצג בצורה מקוצרת בטבלה 11–3:

```
SELECT tsql_code, duration, cs
FROM dbo.Workload
```

טבלה 11–3: תוכן טבלת Workload

tsql_code	*duration*	*cs*
...		
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '20060118';	36094	-184235228
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '20060828';	32662	-184235228
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060101' AND orderdate < '20060201';	717757	368623506
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060401' AND orderdate < '20060501';	684754	368623506
...		

בנקודה זו, תרצה לצבור את הנתונים לפי ה–checksum של חתימת השאילתה. יהיה זה גם שימושי מאוד לקבל צבירות נעות של האחוז של משך כל חתימה ממשך הזמן הכללי. מידע זה יסייע לך לבודד בקלות את תבניות השאילתה אותן עליך לכוונן. זכור שעומסי תפעול טיפוסיים עשויים לכלול מספר גדול של חתימות שאילתה. יהיה הגיוני למלא טבלה זמנית בנתוני הצבירה ולהוסיף לה אינדקס, ואז להריץ שאילתה על הטבלה הזמנית לחישוב הצבירות הנעות.

הרץ את הקוד הבא למילוי הטבלה הזמנית #AggQuries עם סך משך הזמן לכל checksum של חתימה, כולל אחוז מסך–הכל, ומספר שורה המבוסס על משך הזמן בסדר יורד:

```
IF OBJECT_ID('tempdb..#AggQueries') IS NOT NULL
  DROP TABLE #AggQueries;
GO

SELECT cs, SUM(duration) AS total_duration,
  100. * SUM(duration) / SUM(SUM(duration)) OVER() AS pct,
  ROW_NUMBER() OVER(ORDER BY SUM(duration) DESC) AS rn
INTO #AggQueries
FROM dbo.Workload
GROUP BY cs;

CREATE CLUSTERED INDEX idx_cl_cs ON #AggQueries(cs);
```

הרץ את הקוד הבא, המפיק את הפלט המוצג בטבלה 12–3, להחזרת תוכן הטבלה הזמנית:

```
SELECT cs, total_duration, pct, rn
FROM #AggQueries
ORDER BY rn;
```

טבלה 12–3: צבירת Duration עבור checksum של חתימת שאילתה

cs	*total_duration*	*pct*	*rn*
368623506	60723155	98.872121791489952	1
-184235228	660951	1.076189598024132	2
-1872968693	18881	0.030742877762941	3
1018047893	12857	0.020934335013936	4
1037912028	7	0.000011397709037	5

השתמש בשאילתה הבאה להחזרת הצבירות הנעות של האחוזים, תוך סינון רק של השורות בהן האחוז הנע מצטבר לסף מסוים שאתה מגדיר:

```
SELECT AQ1.cs,
  CAST(AQ1.total_duration / 1000.
    AS DECIMAL(12, 2)) AS total_s,
  CAST(SUM(AQ2.total_duration) / 1000.
    AS DECIMAL(12, 2)) AS running_total_s,
  CAST(AQ1.pct AS DECIMAL(12, 2)) AS pct,
  CAST(SUM(AQ2.pct) AS DECIMAL(12, 2)) AS run_pct,
  AQ1.rn
FROM #AggQueries AS AQ1
  JOIN #AggQueries AS AQ2
    ON AQ2.rn <= AQ1.rn
GROUP BY AQ1.cs, AQ1.total_duration, AQ1.pct, AQ1.rn
HAVING SUM(AQ2.pct) - AQ1.pct <= 90 -- percentage threshold
--  OR AQ1.rn <= 5
ORDER BY AQ1.rn;
```

במקרה שלנו, אם אתה משתמש ב–90 אחוז כרמת הסף, תקבל שורה אחת בלבד. למטרות הדגמה, אני השמטתי את חלקו של הביטוי בפסוקית HAVING כדי להחזיר לפחות 5 שורות וקיבלתי את הפלט המוצג בטבלה 3–13.

טבלה 3–13: Duration מצטבר נע

cs	*total_s*	*running_total_s*	*pct*	*running_pct*	*rn*
368623506	60723.16	60723.16	98.87	98.87	1
-184235228	660.95	61384.11	1.08	99.95	2
-1872968693	18.88	61402.99	0.03	99.98	3
1018047893	12.86	61415.84	0.02	100.00	4
1037912028	0.01	61415.85	0.00	100.00	5

ניתן לראות בחלק העליון שישנה תבנית שאילתה אחת שאחראית ל–98.87 אחוז מסך משך הזמן. מניסיוני, לרוב ניתן לספור על יד אחת את כמות תבניות השאילתה הגורמות לרוב בעיות הביצועים במערכת נתונה.

כדי לקבל חזרה את השאילתות עצמן אותן עליך לכוונן, עליך לאחד את טבלת התוצאה המוחזרת מהשאילתה הקודמת עם טבלת Workload, בהתבסס על התאמה של ערך ה–checksum (טור cs), כלהלן:

```
WITH RunningTotals AS
(
  SELECT AQ1.cs,
    CAST(AQ1.total_duration / 1000.
      AS DECIMAL(12, 2)) AS total_s,
    CAST(SUM(AQ2.total_duration) / 1000.
      AS DECIMAL(12, 2)) AS running_total_s,
    CAST(AQ1.pct AS DECIMAL(12, 2)) AS pct,
    CAST(SUM(AQ2.pct) AS DECIMAL(12, 2)) AS run_pct,
    AQ1.rn
  FROM #AggQueries AS AQ1
    JOIN #AggQueries AS AQ2
      ON AQ2.rn <= AQ1.rn
  GROUP BY AQ1.cs, AQ1.total_duration, AQ1.pct, AQ1.rn
  HAVING SUM(AQ2.pct) - AQ1.pct <= 90 -- percentage threshold
--  OR AQ1.rn <= 5
)
SELECT RT.rn, RT.pct, W.tsql_code
FROM RunningTotals AS RT
  JOIN dbo.Workload AS W
    ON W.cs = RT.cs
ORDER BY RT.rn;
```

תקבל את הפלט המוצג בצורה מקוצרת בטבלה 3–14.

טבלה 3–14: השאילתות האיטיות ביותר בצורה מקוצרת

rn	*pct*	*tsql_code*
1	98.87	SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060101' AND orderdate < '20060201';
1	98.87	SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060401' AND orderdate < '20060501';
1	98.87	SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060201' AND orderdate < '20070301';
...		

כמובן, בעומס מציאותי יותר אתה עשוי לקבל חזרה כמות גדולה של שאילתות, אך אתה מעוניין באמת בתבנית השאילתה אותה עליך לכוונן. אז במקום לצרף את טבלת Workload, השתמש באופרטור APPLY כדי להחזיר שורה אחת בלבד לכל חתימת שאילתה עם תבנית השאילתה, ודוגמה אחת לכל תבנית מתוך השאילתות עצמן כלהלן:

```
WITH RunningTotals AS
(
  SELECT AQ1.cs,
    CAST(AQ1.total_duration / 1000.
      AS DECIMAL(12, 2)) AS total_s,
    CAST(SUM(AQ2.total_duration) / 1000.
      AS DECIMAL(12, 2)) AS running_total_s,
    CAST(AQ1.pct AS DECIMAL(12, 2)) AS pct,
    CAST(SUM(AQ2.pct) AS DECIMAL(12, 2)) AS run_pct,
    AQ1.rn
  FROM #AggQueries AS AQ1
    JOIN #AggQueries AS AQ2
      ON AQ2.rn <= AQ1.rn
  GROUP BY AQ1.cs, AQ1.total_duration, AQ1.pct, AQ1.rn
  HAVING SUM(AQ2.pct) - AQ1.pct <= 90 -- percentage threshold
)
SELECT RT.rn, RT.pct, S.sig, S.tsql_code AS sample_query
FROM RunningTotals AS RT
  CROSS APPLY
    (SELECT TOP(1) tsql_code, dbo.fn_SQLSigCLR(tsql_code) AS sig
     FROM dbo.Workload AS W
     WHERE W.cs = RT.cs) AS S
ORDER BY RT.rn;
```

תקבל את הפלט המוצג בטבלה 15–3.

טבלה 15–3: חתימה ודוגמה של השאילתות האיטיות ביותר

rn	*pct*	*sig*	*sample_query*
1	98.87	SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '#' AND orderdate < '#';	SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060101' AND orderdate < '20060201';

כעת באפשרותך למקד את מאמצי הכוונון שלך על תבניות השאילתה שקיבלת חזרה – במקרה שלנו, אחד בלבד. כמובן, בצורה דומה תוכל לזהות את תבניות השאילתה המייצרות את סטי התוצאה הגדולים ביותר, רוב ה–I/O, וכו'.

כוונון אינדקסים/שאילתות

כעת כשאתה יודע אילו תבניות עליך לכוונן, באפשרותך להתחיל בתהליך כוונון–שאילתות ממוקד יותר. התהליך עשוי לערב כוונון אינדקסים או שינויים בקוד שאילתות, ונתרגל זאת בצורה מעמיקה לאורך הספר. ייתכן גם כי תגלה שהשאילתות מכוונות כבר בצורה טובה למדי, ובמקרה כזה תצטרך לבחון היבטים אחרים של המערכת (למשל, חומרה, מבנה מסד הנתונים, וכו'). במקרה שלנו, תהליך הכוונון פשוט למדי. עליך ליצור אינדקס–clustered על הטור orderdate:

```
CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);
```

בהמשך הפרק אציג את נושא כוונון אינדקסים ואסביר מדוע אינדקס–clustered מתאים לתבניות שאילתה כמו אלו שתהליך הכוונון שלנו בודד.

כדי לראות את ההשפעה של הוספת האינדקס, הרץ את הקוד הבא להתחלת trace חדש:

```
DECLARE @dbid AS INT, @traceid AS INT;
SET @dbid = DB_ID('Performance');

EXEC dbo.sp_perfworkload_trace_start
  @dbid     = @dbid,
  @tracefile = 'c:\temp\Perfworkload 20060829',
  @traceid  = @traceid OUTPUT;
```

כאשר הרצתי את הקוד הזה, קיבלתי את הפלט הבא המראה שקוד ה–trace שנוצר הוא 2:

```
Trace ID: 2, Trace File: 'c:\temp\Perfworkload 20060829.trc'
```

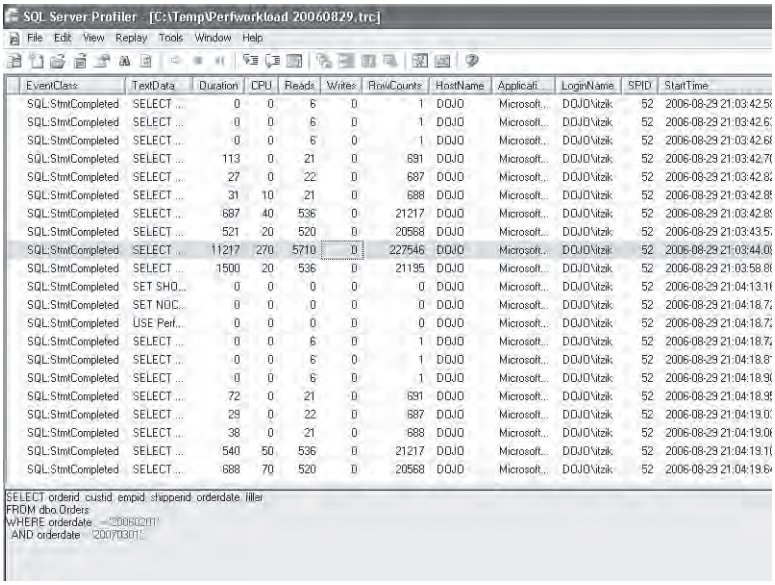
הרץ את השאילתות לדוגמה מקטע–קוד 2–3 שוב, ואז עצור את ה–trace:

```
EXEC sp_trace_setstatus 2, 0;
EXEC sp_trace_setstatus 2, 2;
```

תרשים 5–3 מציג את נתוני ה–trace שנטענו עם Profiler.

ניתן לראות שמשך הזמן וה–I/O המעורבים בתבנית השאילתה שכוונן פחתו משמעותית. עדיין, ישנן שאילתות המייצרות תנועה רבה ברשת. עם אלו, ייתכן שתרצה לבדוק האם חלק מעיבוד התוצאות שלהם יכול להיות מושג בצד השרת, וכך להפחית את כמות הנתונים הנשלחת ברשת.

תרשים 5–3: נתוני ה–trace של העומס ב–Performance לאחר הוספת אינדקס



כלים לכוונון שאילתות

סעיף זה מספק סקירה של כלים לכוונון–שאילתות בהם נשתמש לאורך הספרים, והוא יתמקד בניתוח תוכניות עבודה.

Syscacheobjects

SQL Server 2000 מספק לך טבלת מערכת וירטואלית הנקראת master.dbo.syscacheobjects, המכילה מידע על תוכניות עבודה ב–cache. SQL Server 2005 מספק לך view לתאימות אחורנית הנקרא sys.syscacheobjects, ו–DMV ושני DMFs חדשים המחליפים את syscacheobjects. שלושת האובייקטים החדשים הם: sys.dm_exec_cached_plans, sys.dm_exec_plan_attributes ו– sys.dm_exec_sql_text. אלו נותנים לך מידע שימושי ביותר כאשר מנתחים את התנהגות תוכניות העבודה ושימוש ב–cache, קומפילציות וקומפילציות–חוזרות. האובייקט sys. dm_exec_cached_plans מכיל מידע על תוכניות העבודה ב– cache; sys.dm_exec_plan_attributes מכיל שורה אחת לכל תכונה הקשורה לתוכנית, אשר ה–handle שלה מסופק כקלט ל– DMF; sys.dm_exec_sql_text מחזיר את הקוד הקשור לשאילתה, אשר ה–handle שלה מסופק כקלט ל–DMF.

ניקוי ה–Cache

כאשר מנתחים ביצועי שאילתה, לעיתים עליך לנקות את ה–cache. SQL Server מספק לך כלים לנקות הן נתונים והן תוכניות עבודה מה–cache. כדי לנקות נתונים מה–cache גלובלית, השתמש בפקודה הבאה:

```
DBCC DROPCLEANBUFFERS;
```

לניקוי תוכניות עבודה מה–cache גלובלית, השתמש בפקודה הבאה:

```
DBCC FREEPROCCACHE
```

לניקוי תוכניות עבודה של מסד נתונים מסוים, השתמש בפקודה הבאה:

```
DBCC FLUSHPROCINDB(<db_id>);
```

שים לב שהפקודה DBCC FLUSHPROCINDB אינה מתועדת.

אזהרה: היזהר מפני שימוש בפקודות אלו במערכות תפעוליות משום שכמובן, לניקוי ה–cache תהיה השפעה על המערכת כולה. לאחר ניקוי הנתונים מה–cache, SQL Server יצטרך לקרוא פיסית מהדיסק דפים אליהם הוא ניגש בפעם הראשונה. לאחר ניקוי תוכניות עבודה מה–cache, SQL Server יצטרך לייצר תוכניות עבודה חדשות לשאילתות. כמו כן, ודא שאתה מודע להשפעה הגלובלית של ניקוי ה–cache גם כאשר אתה מבצע זאת בסביבות פיתוח או בדיקות.

אובייקטי ניהול דינמיים (Dynamic Management Objects)

SQL Server 2005 מציג למעלה מ–70 אובייקטי ניהול דינמיים, כולל DMVs ו–DMFs. אלו מכילים מידע שימושי ביותר על השרת שבאפשרותך להשתמש כדי לנטר את ה– SQL Server, לאבחן בעיות ולכוונן ביצועים. חלק גדול מהמידע המסופק על ידי views ועל ידי פונקציות אלו מעולם לא היה זמין בעבר. כדאי מאוד להשקיע זמן בלימוד מעמיק שלהם. בספרים אלו, אשתמש באובייקטים הרלוונטיים לדיון שלי, אך אני מעודד אותך לבחון מקרוב גם אחרים. תוכל למצוא מידע אודותם ב– Books Online, ואתן לך גם מקורות מידע נוספים בסוף הפרק.

STATISTICS IO

STATISTICS IO היא אפשרות session הנמצאת בשימוש רב בספרים אלו. היא מחזירה מידע I/O לגבי המשפטים שאתה מריץ. כדי להדגים את השימוש בה, ראשית נקה את cache הנתונים:

```
DBCC DROPCLEANBUFFERS;
```

כעת הרץ את הקוד הבא כדי לשנות את מצב אפשרות ה–session ל–on ולקרוא לשאילתה:

```
SET STATISTICS IO ON;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060101'
  AND orderdate < '20060201';
```

אתה אמור לקבל פלט דומה לזה:

```
Table 'Orders'. Scan count 1, logical reads 536, physical reads 2, read-ahead
reads 532, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

הפלט אומר לך כמה פעמים בוצעה גישה לטבלה בתוכנית (Scan count); כמה קריאות מה–cache היו מעורבות (logical reads); כמה קריאות מהדיסק היו מעורבות (physical reads) ו– (read-ahead reads); ובדומה, כמה קריאות לוגיות ופיסיות הקשורות לאובייקטים גדולים היו מעורבות (lob logical reads, lob physical reads, lob read-ahead reads).

הרץ את הקוד הבא כדי לשנות את מצב אפשרות ה–session ל–off:

```
SET STATISTICS IO OFF;
```

מדידת זמן הריצה של שאילתות

STATISTICS TIME היא אפשרות session המחזירה את מידע זמן המעבד נטו וזמן השעון לגבי המשפטים שאתה מריץ. היא מחזירה מידע זה הן עבור הזמן שלקח לפרק ולבצע קומפילציה לשאילתה, והן עבור הזמן שלקח להריץ אותה. להדגמת השימוש באפשרות session זו, ראשית נקה מה–cache הן את הנתונים והן את תוכניות העבודה:

```
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
```

הרץ את הקוד הבא כדי לשנות את מצב אפשרות ה–session ל–on:

```
SET STATISTICS TIME ON;
```

כעת קרא לשאילתה הבאה:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060101'
  AND orderdate < '20060201';
```

תקבל פלט דומה לזה:

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 30 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

SQL Server Execution Times:
   CPU time = 30 ms,  elapsed time = 619 ms.
```

הפלט נותן לך את זמן המעבד נטו ואת זמן השעון לפירוק וקומפילציה של השאילתה, וכן את הזמן שלקח להריץ אותה. הרץ את הקוד הבא כדי לשנות את מצב אפשרות ה–session ל–off:

```
SET STATISTICS TIME OFF;
```

כלי זה נוח כאשר ברצונך לנתח את הביצועים של שאילתה בודדת בצורה אינטראקטיבית. כאשר אתה מריץ בוחני ביצועים ב–batch, הדרך למדוד את זמן הריצה של שאילתות שונה. אחסן את הערך של הפונקציה GETDATE במשתנה מייד לפני השאילתה. מייד לאחר השאילתה, הרץ משפט INSERT לתוך הטבלה בה אתה אוסף את מידע הביצועים, תוך הפחתת הערך המאוחסן במשתנה מהערך הנוכחי של GETDATE. שים לב ש–GETDATE מחזיר ערך DATETIME, בעל מידת דיוק של 3.33 אלפיות שנייה. כאשר מודדים את סטטיסטיקות הזמן של שאילתות עבורן רמת דיוק זו אינה מספיקה, הרץ את השאילתה בלולאה וחלק את זמן הריצה של הלולאה כולה במספר ההרצות.

ניתוח תוכניות עבודה (Execution Plans)

תוכנית עבודה מיוצרת על ידי ה–optimizer כדי להחליט כיצד לעבד שאילתה נתונה. התוכנית מכילה אופרטורים המופעלים כללית בסדר מסוים. ישנם אופרטורים שיכולים להיות מופעלים כאשר האופרטור הקודם עדיין בתהליך. ישנם אופרטורים היכולים להיות מופעלים יותר מפעם אחת. כמו כן, ישנם ענפים של התוכנית המופעלים בצורה מקבילית אם ה–optimizer בחר תוכנית מקבילית. בתוכנית, ה–optimizer קובע את סדר הגישה לטבלאות המעורבות בשאילתה, באילו אינדקסים להשתמש ובאילו שיטות גישה להשתמש עבורם, באילו אלגוריתמים של join להשתמש, וכו'. למעשה, עבור שאילתה נתונה ה–optimizer שוקל מספר תוכניות עבודה, והוא בוחר בתוכנית בעלת העלות הנמוכה ביותר מבין אלו שנוצרו. שים לב שייתכן כי SQL Server לא ייצר את כל תוכניות העבודה האפשריות לשאילתה נתונה. אם היה עושה כך תמיד, תהליך האופטימיזציה עלול היה לארוך זמן רב מדי. SQL Server יחשב עלות סף לתהליך האופטימיזציה, בין היתר בהתבסס על גודל הטבלאות המעורבות בשאילתה. בסוף הפרק אכוון אותך למאמר המספק מידע מפורט לגבי תהליך זה.

לאורך ספרים אלו, אנתח לעיתים קרובות תוכניות עבודה של שאילתות. סעיף זה והסעיף הבא ("כוונון אינדקסים") ייתנו לך את הרקע הנדרש לצורך מעקב והבנה של הדיונים הנוגעים לניתוח תוכניות. שים לב שהמטרה של סעיף זה אינה לגרום לך להכיר את כל האופרטורים האפשריים, אלא להכיר לך את הטכניקות לניתוח תוכניות. הסעיף "כוונון אינדקסים" יכיר לך אופרטורים הקשורים באינדקסים, ומאוחר יותר בספר ארחיב בנוגע לאופרטורים נוספים – למשל, אופרטורים הקשורים ל–join יתוארו בפרק 5.

תוכניות עבודה גרפיות

לאורך ספרים אלו קיים שימוש רב בתוכניות עבודה גרפיות. SSMS מאפשר לך לקבל הן תוכנית עבודה משוערת (על ידי הקשה על Ctrl+L) והן תוכנית בפועל (על ידי הקשה על Ctrl+M) בנוסף לפלט השאילתה שאתה מריץ. שים לב ששתיהן יתנו לך את אותה תוכנית; זכור שתוכנית עבודה מיוצרת בטרם השאילתה מורצת. עם זאת, כאשר אתה מבקש תוכנית משוערת, השאילתה אינה מורצת כלל. כמובן שמדידות מסוימות יכולות להיאסף רק בזמן ריצה (למשל, מספר השורות בפועל המוחזרות מכל אופרטור, והמספר בפועל של rebinds ושל rewinds). בתוכנית המשוערת, תראה הערכות למדידות היכולות להיאסף רק בזמן ריצה, בעוד התוכנית בפועל תציג את המדידות בפועל וכן כמה מאותן הערכות.

כדי להדגים ניתוח תוכנית עבודה גרפית, אשתמש בשאילתה הבאה:

```
SELECT custid, empid, shipperid, COUNT(*) AS numorders
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;
```

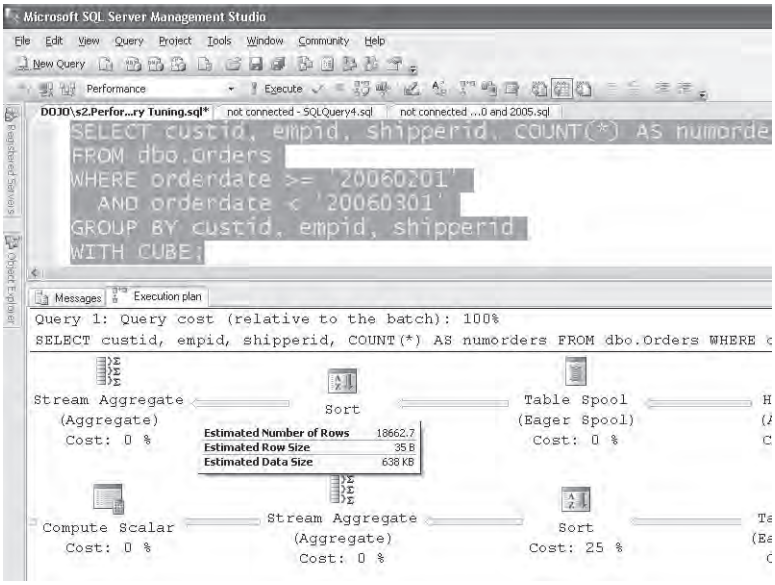
השאילתה מחזירה ספירות מצטברות של הזמנות בתקופה נתונה, מקובצות לפי custid, empid ו–shipperid, כולל החישובים של צבירות על (מיוצרות על ידי אפשרות CUBE). בשאילתות CUBE אדון בפירוט בפרק 6.

שים לב: ביצעתי מספר מניפולציות גרפיות על התוכניות הגרפיות המופיעות בפרק זה, כדי להתאים תרשימים לדפים המודפסים ולשם הבהירות.

כדוגמה, אם אתה מבקש תוכנית עבודה משוערת עבור השאילתה הקודמת, תקבל את התוכנית המוצגת בתרשים 6–3.

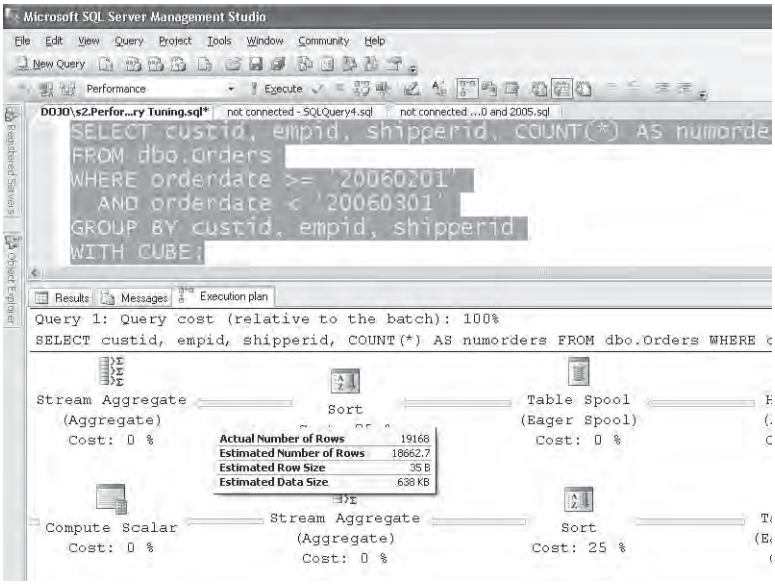
שים לב שכאשר אתה מציב את סמן העכבר שלך מעל חץ היוצא מאופרטור (למשל, זה שיוצא מהאופרטור Sort הראשון), אתה מקבל מספר שורות משוער. דרך אגב, היבט נחמד של החיצים המייצגים זרימת נתונים הוא שהעובי שלהם פרופורציונלי למספר השורות המוחזרות על ידי אופרטור המקור. תרצה לשים לב במיוחד לחיצים עבים, שכן אלו עשויים להעיד על בעיית ביצועים.

תרשים 3–6: דוגמה לתוכנית עבודה משוערת



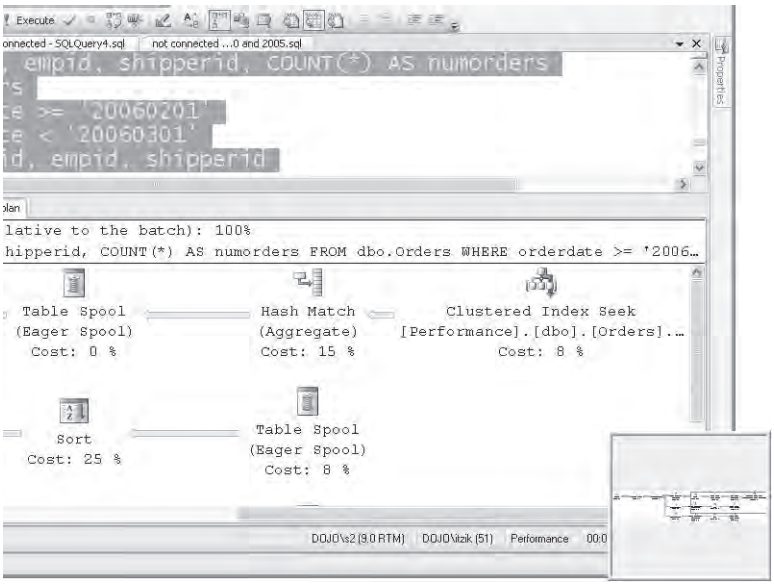
כעת, שנה את מצב האפשרות Include Actual Execution Plan ל–on, והרץ את השאילתה. תקבל הן את הפלט של השאילתה והן את התוכנית בפועל, כפי שנראה בתרשים 3–7.

תרשים 3–7: דוגמה לתוכנית עבודה בפועל



שים לב שכעת אתה מקבל את מספר השורות בפועל המוחזרות על ידי אופרטור המקור. כאשר אתה מקבל תוכניות רחבות כמו זו שאינן מתאימות למסך אחד, תוכל להשתמש בכלי תקריב חדש ונחמד. הקש על מקש + המופיע בפינה הימנית התחתונה של חלון תוכנית העבודה, ותקבל מרובע המאפשר לך לנווט למקום מסוים בתוכנית, כפי שמוצג בתרשים 8–3.

תרשים 8–3: כלי תקריב ב–showplan גרפי



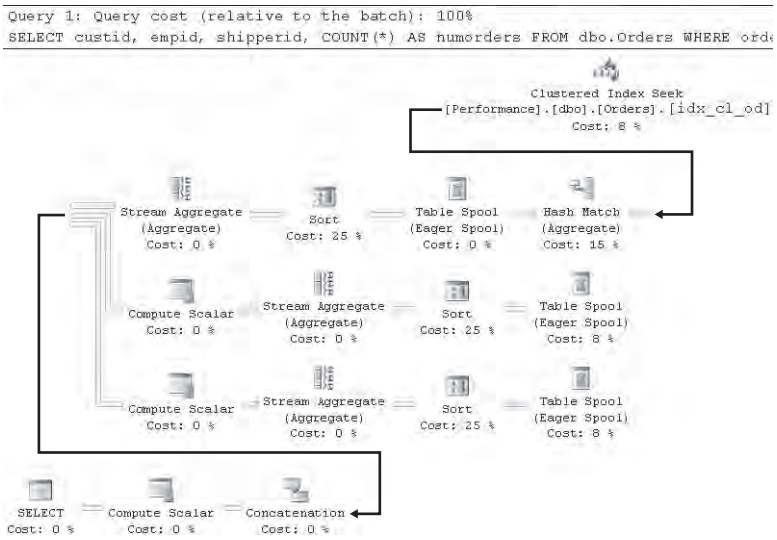
תרשים 9–3 מראה את תוכנית העבודה המלאה לשאילתת ה–CUBE שלנו – זאת לאחר כמה מניפולציות גרפיות שבוצעו לצורך בהירות ולהתאמת התוכנית למסך אחד.

שיניתי את המיקום של כמה מהאופרטורים והוספתי חיצים אדומים כדי לציין את הזרימה המקורית. כמו כן, כללתי את שמות האובייקטים המלאים במקומות הרלוונטיים. בתוכנית המקורית, שמות אובייקטים נקטעים אם הם ארוכים מדי.

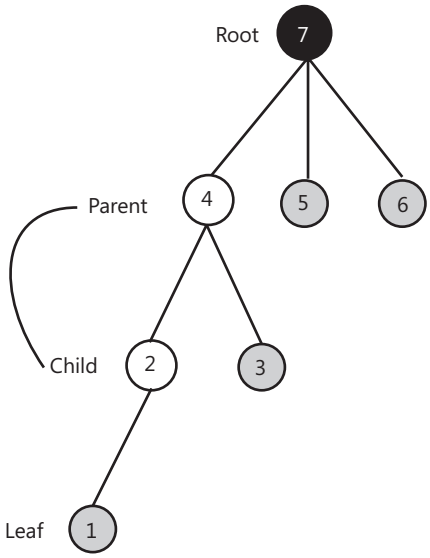
תוכנית היא עץ של אופרטורים. נתונים זורמים מאופרטור ילד לאופרטור אב. סדר העץ של תוכניות גרפיות אותו אתה מקבל ב–SSMS מבוטא מימין לשמאל ומלמעלה למטה. זהו לרוב הסדר בו עליך לנתח תוכנית כדי לבחון את זרימת הפעילות. במקרה שלנו, האופרטור Clustered Index Seek הוא האופרטור הראשון המתחיל את הזרימה, המעביר את הפלט שלו לאופרטור הבא בעץ – Hash Match (Aggregate) – וכך הלאה.

במדעי המחשב, מבני עצים משורטטים לרוב עם אלמנט השורש למעלה והעלים למטה; וקדימות בין אחים משורטטת משמאל לימין. אם אתה מורגל בעבודה עם אלגוריתמים הקשורים בעצים (או גרפים הכלליים יותר), ייתכן שתרגיש נוח עם הייצוג הנפוץ יותר של עצים, כאשר ההפעלה של האופרטורים תזרום בסדר המוצג בתרשים 10–3.

תרשים 3–9: תוכנית עבודה לשאילתת CUBE

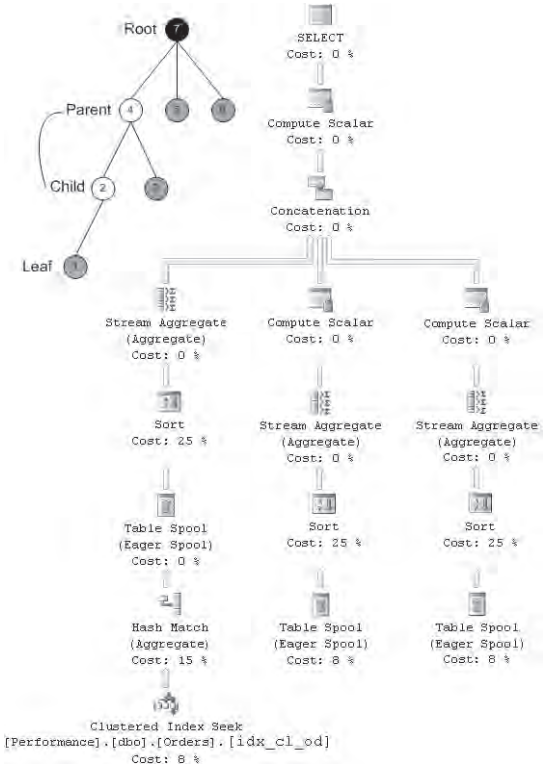


תרשים 3–10: עץ



המספרים בתרשים מייצגים סדר פעולה של אופרטורים בתוכנית. אם אתה מרגיש נוח יותר עם הייצוג הנפוץ במדעי המחשב כפי שנראה בתרשים 3–10, ייתכן שתעריך את השרטוט המוצג בתרשים 3–11 של תוכנית שאילתת ה–CUBE שלנו.

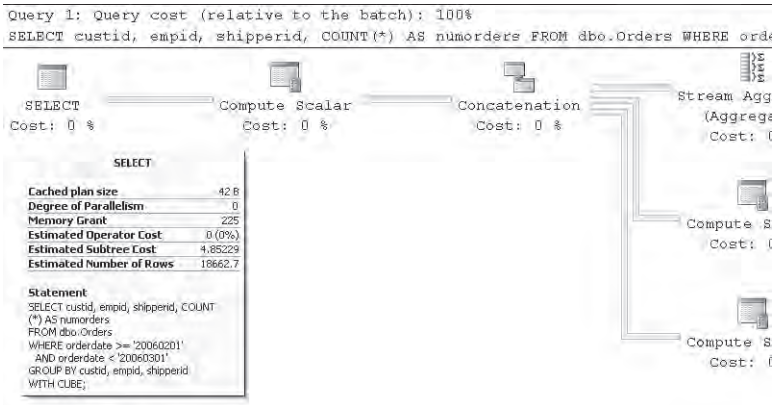
תרשים 11–3: תוכנית עבודה לשאילתת CUBE (מסודרת מחדש)



לוגית, כדי לייצר תוכנית כזו, עליך לסובב את התוכנית המקורית שאתה מקבל מ–SSMS, 90 מעלות לשמאל, ואז להפוך אותה אנכית לצד השני. אם כי, לא נראה לי שתרצה לנסות זאת בבית. לקח לי יום שלם לייצר את התרשים הזה, בעבודה ברמת הפיקסל עם אחד מהכלים המועדפים עלי – mspaint.exe. עשיתי זאת בעיקר כמחווה לידידי והמנטור שלי לובור קולר. לובור, זה בשבילך!

חזור לתוכנית העבודה המקורית עבור שאילתת ה–CUBE המוצגת בתרשים 9–3 כדי לבחון היבטים אחרים של התוכנית. שים לב שישנו אחוז עלות מקושר לכל אופרטור. ערך זה הוא האחוז של עלות האופרטור מכלל עלות השאילתה, כפי שמוערך על ידי ה–optimizer. מייד אסביר מה מאחורי ערך עלות השאילתה. ברצונך לשים לב במיוחד לאופרטורים המערבים ערכי אחוז גבוהים, ולמקד את מאמצי הכוונון שלך באופרטורים אלו. כאשר תמקם את סמן העכבר שלך מעל אופרטור, תקבל תיבת מידע צהובה, שאתאר מייד. אחת מהמידות שתמצא שם נקראת Estimated Subtree Cost. ערך זה מייצג את הערך המצטבר המשוער של תת–העץ, מתחילת האופרטור הנוכחי (כל האופרטורים מכל הענפים המובילים לאופרטור הנוכחי). עלות תת–העץ המקושרת עם אופרטור השורש (עליון–ביותר, שמאלי–ביותר) מייצגת את העלות המשוערת של השאילתה כולה, כפי שמוצג בתרשים 12–3.

תרשים 12–3:
עלות תת–עץ



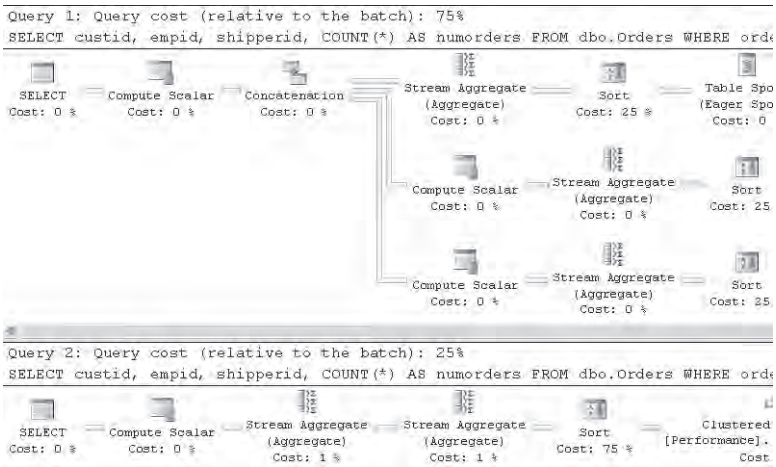
ערך העלות של השאילתה מיוצר על ידי נוסחאות אשר, אם נתבטא בחופשיות, מטרתם לשקף את מספר השניות שלוקח לשאילתה לרוץ על מכונת בדיקות בה השתמשו מפתחי SQL Server במעבדות שלהם. למשל, עלות תת–העץ המשוערת של שאילתת ה–CUBE שלנו היא מעט פחות מ–5 כלומר, שהנוסחאות מעריכות שייקח לשאילתה בערך קרוב ל–5 שניות לרוץ על מכונת הבדיקות של מיקרוסופט. כמובן, זוהי הערכה. ישנם כל כך הרבה גורמים המעורבים באלגוריתמים של העלות שאפילו על מכונת הבדיקות המקורית של מיקרוסופט תראה שינויים גדולים בין זמן הריצה בפועל של שאילתה וערך העלות המשוער שלה. בנוסף, במערכת בה אתה רץ, החומרה ומבנה מסד הנתונים שלך עשויים להיות שונים משמעותית מהמערכת בה השתמשה מיקרוסופט לכיול. לפיכך, אל לך לצפות להתאמה מלאה בין עלות תת–העץ של שאילתה לבין זמן הריצה שלה בפועל.

דבר נוסף נחמד בתוכנית העבודה הגרפית הוא שאתה יכול להשוות בקלות עלויות של מספר שאילתות. ייתכן שתרצה להשוות את העלויות של שאילתות שונות המייצרות את אותה תוצאה, או במקרים מסוימים אפילו שאילתות שעושות דברים שונים מעט. למשל, נניח שברצונך להשוות את עלות השאילתה שלנו המשתמשת באפשרות CUBE לשאילתה דומה המשתמשת באפשרות ROLLUP:

```
SELECT custid, empid, shipperid, COUNT(*) AS numorders
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;

SELECT custid, empid, shipperid, COUNT(*) AS numorders
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH ROLLUP;
```

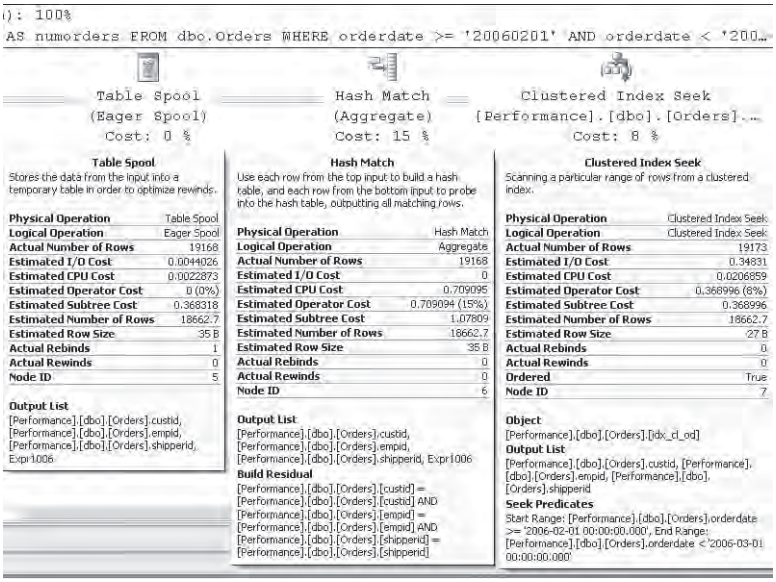
אתה מסמן את השאילתות שברצונך להשוות ומבקש תוכנית עבודה גרפית (משוערת או בפועל, כפי שנדרש). במקרה שלנו, תקבל את התוכניות המוצגות בתרשים 13–3.

תרשים 13–3: השוואת עלויות של תוכניות עבודה



בראש כל תוכנית, תקבל את האחוז של עלות השאילתה המשוערת מתוך כלל ה–batch. למשל, במקרה שלנו, תוכל לראות ששאילתת ה–CUBE מוערכת כיקרה פי שלוש מאשר שאילתת ROLLUP. כאשר תמקם את סמן העכבר שלך מעל אופרטור, תקבל תיבת ToolTip צהובה המכילה מידע על האופרטור, כפי שמוצג בתרשים 14–3.

תרשים 14–3: תיבת ToolTip למידע על אופרטור

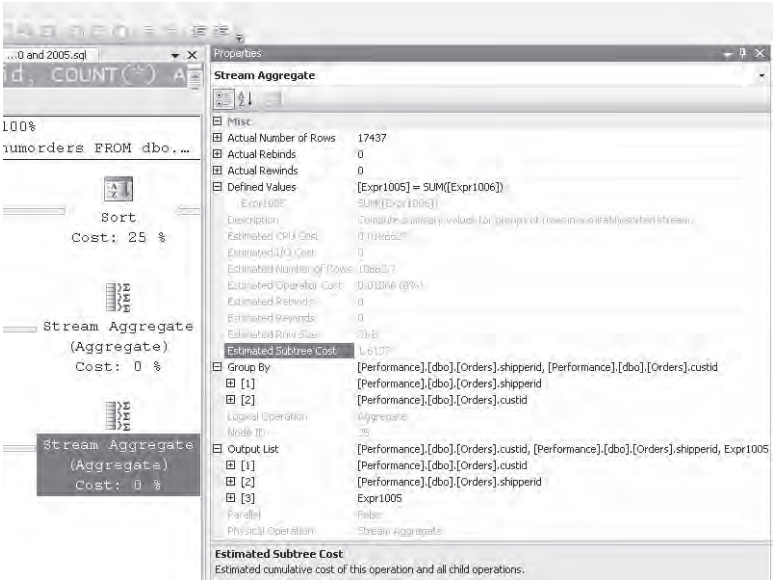


תיבת המידע תספק לך את המידע הבא:

- שם האופרטור ותיאור קצר של הפונקציה שלו.
- **Physical Operation**: הפעולה הפיסית שתתרחש במנוע.
- **Logical Operation**: הפעולה הלוגית לפי המודל התפיסתי של מיקרוסופט של עיבוד שאילתה. למשל, עבור אופרטור join תקבל את אלגוריתם ה–join המשמש כפעולה הפיסית (Nested Loops, Merge, Hash), וסוג ה–join הלוגי המשמש כפעולה הלוגית (Inner Join, Outer Join, Semi Join, וכו'). כאשר לא קיימת פעולה לוגית המקושרת לאופרטור, מידה זו תראה ערך זהה לזה שבפעולה הפיסית.
- **Actual Number of Rows**: מספר השורות המוחזרות בפועל מהאופרטור (מוצג רק עבור תוכניות בפועל).
- **Estimated I/O Cost** ו– **Estimated CPU Cost**: החלק המשוער של עלות האופרטור הקשורה למשאב המסוים (I/O או מעבד). מידות אלו יסייעו לך לזהות האם האופרטור מנצל יותר משאבי I/O או מעבד. למשל, תוכל לראות שהאופרטור Clustered Index Seek כרוך בעיקר ב–I/O, בעוד האופרטור Hash Match כרוך בעיקר במעבד.
- **Estimated Operator Cost**: העלות הקשורה באופרטור המסוים.
- **Estimated Subtree Cost**: כפי שתואר קודם, העלות המשוערת הקשורה בתת–העץ כולו עד לצומת הנוכחית.
- **Estimated Number of Rows**: המספר המשוער של שורות שיוחזרו מאופרטור זה. במקרים מסוימים, תוכל לזהות בעיות עלות הקשורות לסטטיסטיקות לא מספיקות או לסיבות אחרות על ידי בחינת אי–התאמות בין מספר השורות בפועל והמספר המשוער.
- **Estimated Row Size**: ייתכן שתשאל את עצמך מדוע ערך בפועל של מדד זה אינו מוצג בתוכנית השאילתה בפועל. הסיבה היא שייתכן שיש לך טיפוסי טורים בעלי אורך–דינמי בטבלה שלך עם שורות שאורכן משתנה.
- **Actual Rebinds** ו– **Actual Rewinds**: מדדים אלו רלוונטיים רק לאופרטורים המופיעים בצד הפנימי של Nested Loops join; אחרת, Rebinds יראו 1 ו–Rewinds יראו 0. אופרטורים אלו מתייחסים למספר הפעמים שנקראת שיטת Init פנימית. סכום מספר ה–Rebinds וה–Rewinds צריך להיות שווה למספר השורות המעובדות בצד החיצוני של ה–join. משמעותו של rebind היא שאחד או יותר מהפרמטרים הקשורים ל–join השתנה והצד הפנימי חייב להיות מוערך מחדש. משמעותו של rewind היא שאף אחד מהפרמטרים הקשורים ל–join לא השתנה וסט התוצאה הפנימי הקודם ניתן לשימוש מחדש.
- חלק תחתון של תיבת המידע: מציג היבטים אחרים הקשורים לאופרטור, כמו שם האובייקט הקשור אליו, פלט, ארגומנטים, וכו'.

ב– SQL Server 2005 אתה יכול לקבל כיסוי מפורט יותר של מאפייני האופרטור בחלון Properties (על ידי הקשה על F4), כפי שמוצג בתרשים 15–3.

תרשים 15–3:
חלון Properties



כיסוי של תוכניות עבודה גרפיות ימשיך בסעיף "כוונון אינדקסים" כאשר נדון בשיטות גישה לאינדקס.

Showplan טקסטואלי

SQL Server נותן לך גם כלים לקבלת תוכנית עבודה בפורמט טקסטואלי. למשל, אם תשנה את מצב אפשרות ה– SHOWPLAN_TEXT session ל–on, כאשר תריץ שאילתה, SQL Server לא יעבד אותה. במקום זאת הוא רק ייצר תוכנית עבודה ויחזיר אותה כטקסט. להדגמת אפשרות session זו, שנה את מצבה ל–on על ידי הרצת הקוד הבא:

```
SET SHOWPLAN_TEXT ON;
```

כעת קרא לשאילתה הבאה:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 280885;
```

תקבל את הפלט הבא:

```
  |--Nested Loops(Inner Join, OUTER REFERENCES:([Uniq1002],
       [Performance].[dbo].[Orders].[orderdate]))
```

```
    |--Index Seek(OBJECT:([Performance].[dbo].[Orders].[PK_Orders]),
         SEEK:([Performance].[dbo].[Orders].[orderid]=[@1])
           ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:(
         [Performance].[dbo].[Orders].[idx_cl_od]),
         SEEK:([Performance].[dbo].[Orders].[orderdate]=
         [Performance].[dbo].[Orders].[orderdate]
         AND [Uniq1002]=[Uniq1002]) LOOKUP ORDERED FORWARD)
```

כדי לנתח את השאילתה, אתה "קורא" או "עוקב" אחר הענפים ברמות הפנימיות לפני אלו ברמות החיצוניות (מלמטה למעלה), וענפים המופיעים באותה רמה מלמעלה למטה. כפי שתוכל לראות, אתה מקבל רק את שמות האופרטורים ואת הארגומנטים הבסיסיים שלהם. הרץ את הקוד הבא לשנות את מצב אפשרות ה–session ל–off:

```
SET SHOWPLAN_TEXT OFF;
```

אם ברצונך במידע מפורט יותר על התוכנית הדומה למה שתוכנית העבודה הגרפית נותנת לך, השתמש באפשרות ה– SHOWPLAN_ALL session כדי לקבל תוכנית משוערת ובאפשרות ה– STATISTICS PROFILE session כדי לקבל את זו בפועל. SHOWPLAN_ALL יפיק תוצאת טבלה עם המידע המסופק על ידי SHOWPLAN_TEXT, וכן את המדדים הבאים: StmtText, StmtId, NodeId, Parent, PhysicalOp, LogicalOp, Argument, DefinedValues, EstimateRows, EstimateIO, EstimateCPU, AvgRowSize, TotalSubtreeCost, OutputList, Warnings, Type, Parallel ו–EstimateExecusions.

לבדיקת אפשרות session זו, שנה את מצבה ל–on:

```
SET SHOWPLAN_ALL ON;
```

הרץ את השאילתה הקודמת, ובחן את התוצאה. כאשר תסיים, שנה את מצב האפשרות ל–off:

```
SET SHOWPLAN_ALL OFF;
```

האפשרות STATISTICS PROFILE תפיק תוכנית בפועל. השאילתה תרוץ והפלט שלה יופק. תקבל גם את הפלט המוחזר על ידי SHOWPLAN_ALL. בנוסף, תקבל את הטורים Rows ו–Executes, המחזיקים ערכים בפועל בניגוד לאלו המשוערים. לבחינת אפשרות session זו, שנה את מצבה ל–on:

```
SET STATISTICS PROFILE ON;
```

הרץ את השאילתה הקודמת, ובחן את התוצאה. כאשר תסיים, שנה את המצב ל–off:

```
SET STATISTICS PROFILE OFF;
```

XML Showplans

אם ברצונך לפתח קוד משל עצמך שיפרק וינתח מידע תוכנית עבודה, תמצא את המידע המוחזר על ידי SHOWPLAN_TEXT, SHOWPLAN_ALL ו– STATISTICS PROFILE קשה מאוד לעבודה. SQL Server 2005 מציג שתי אפשרויות session חדשות המאפשרות לך לקבל מידע תוכנית עבודה משוערת ובפועל במבנה XML; נתוני XML נוחים הרבה יותר לפירוק ועבודה על ידי קוד יישום. אפשרות ה– SHOWPLAN_XML session תפיק ערך XML עם מידע התוכנית המשוערת, ואפשרות ה– STATISTICS XML session תפיק ערך עם מידע תוכנית בפועל. כדי לבחון את SHOWPLAN_XML, שנה את מצבה ל–on על ידי הרצת הקוד הבא:

```
SET SHOWPLAN_XML ON;
```

כעת הרץ את השאילתה הבאה:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 280885;
```

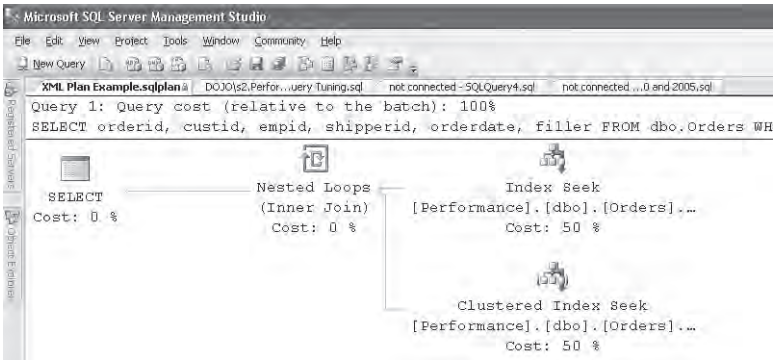
תקבל את ערך ה–XML הבא, המוצג כאן בצורה מקוצרת:

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan" Version="1.0" Build="9.00.1399.06">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT orderid, custid, empid,
shipperid, orderdate, filler&#xD;&#xA;FROM dbo.Orders&#xD;&#xA;WHERE orderid
= 280885;" StatementId="1" StatementCompId="1" StatementType="SELECT"
StatementSubTreeCost="0.00657038" StatementEstRows="1" StatementOptmLev
el="TRIVIAL">
          <StatementSetOptions QUOTED_IDENTIFIER="false"
ARITHABORT="true" CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false"
ANSI_PADDING="false" ANSI_WARNINGS="false" NUMERIC_ROUNDABORT="false" />
          <QueryPlan CachedPlanSize="14">
            <RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner
Join" EstimateRows="1" EstimateIO="0" EstimateCPU="4.18e-006"
AvgRowSize="195" EstimatedTotalSubtreeCost="0.00657038" Parallel="0"
EstimateRebinds="0" EstimateRewinds="0">
              <OutputList>
                <ColumnReference Database="[Performance]"
Schema="[dbo]" Table="[Orders]" Column="orderid" />
                <ColumnReference Database="[Performance]" Schema="[dbo]"
```

```
Table="[Orders]" Column="custid" />
                <ColumnReference Database="[Performance]"
Schema="[dbo]" Table="[Orders]" Column="empid" />
                <ColumnReference Database="[Performance]"
Schema="[dbo]" Table="[Orders]" Column="shipperid" />
                <ColumnReference Database="[Performance]"
Schema="[dbo]" Table="[Orders]" Column="orderdate" />
                <ColumnReference Database="[Performance]"
Schema="[dbo]" Table="[Orders]" Column="filler" />
              </OutputList>
              <NestedLoops Optimized="0">
                <OuterReferences>
                  <ColumnReference Column="Uniq1002" />
                  <ColumnReference Database="[Performance]"
Schema="[dbo]" Table="[Orders]" Column="orderdate" />
                </OuterReferences>
                <RelOp NodeId="1" PhysicalOp="Index Seek"
LogicalOp="Index Seek" EstimateRows="1" EstimateIO="0.003125"
EstimateCPU="0.0001581" AvgRowSize="23" EstimatedTotalSubtreeCost="0.0
032831" Parallel="0" EstimateRebinds="0" EstimateRewinds="0">
...
          </QueryPlan>
        </StmtSimple>
      </Statements>
    </Batch>
  </BatchSequence>
</ShowPlanXML>
```

שים לב שאם תשמור את ערך ה–XML בקובץ עם הסיומת sqlplan., תוכל אז לפתוח אותו עם SSMS ולקבל תצוגה גרפית של תוכנית העבודה, כפי שמוצג בתרשים 16–3.

תרשים 16–3: דוגמת תוכנית XML



הרץ את הקוד הבא כדי לשנות את מצב אפשרות ה–session ל–off:

```
SET SHOWPLAN_XML OFF;
```

כפי שציינתי קודם, כדי לקבל ערך XML עם מידע לגבי תוכנית העבודה בפועל, השתמש באפשרות ה–STATISTICS XML session כלהלן:

```
SET STATISTICS XML ON;
GO
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 280885;
GO
SET STATISTICS XML OFF;
```

Hints

Hints מאפשרים לך לדרוס את התנהגות ברירת המחדל של SQL Server בהיבטים שונים, ו– SQL Server יפעל לפי דרישתך כאשר הדבר אפשרי טכנית. המושג hint מטעה משום שהוא אינו מחווה של רצון טוב אותה SQL Server יכול לבצע או שלא; אלא, אתה כופה על SQL Server התנהגות מסוימת כאשר הדבר אפשרי טכנית. מבחינה תחבירית, קיימים שלושה סוגי hints: join hints, hints של שאילתה ו–hints של טבלה. Join hints מצוינים בין מילת המפתח המייצגת את סוג ה–join ומילת המפתח JOIN (למשל, INNER MERGE JOIN). Hints של שאילתה מצוינים בפסוקית OPTION לאחר השאילתה עצמה; למשל, SELECT … OPTION (OPTIMIZE FOR (@od = '99991231')). Hints של טבלאות מצוינים מייד לאחר שם טבלה או כינוי בפסוקית WITH (למשל FROM dbo.Orders WITH (index = idx_unc_od_oid_i_cid_eid)).

Hints יכולים להיות מסווגים בקטגוריות שונות בהתבסס על הפונקציונליות שלהם, כולל: hints של אינדקס, join hints, פרלליות, נעילות, קומפילציה ואחרים. ישנם hints הקשורים לביצועים, כמו חיוב שימוש באינדקס מסוים, אשר להם היבטים הן חיוביים והן שליליים. בצד השלילי, hint הופך את ההיבט המסוים של התוכנית לסטטי. כאשר משתנים הנתונים הנמצאים בטבלאות עליהן מבוצעת השאילתה, ה–optimizer לא ייוועץ בסטטיסטיקות לקביעה האם כדאי להשתמש באינדקס או לא, משום שאתה מכריח אותו להשתמש בו תמיד. אתה מאבד את היתרון שבאופטימיזציה מבוססת–עלות שה–optimizer של SQL Server מספק לך. בצד החיובי, על ידי הגדרת hints אתה מפחית את הזמן שלוקח לבצע אופטימיזציה לשאילתות. במקרים מסוימים, אתה דורס בחירות לא יעילות שה–optimizer מבצע לעיתים כתוצאה מהטבע של אופטימיזציה מבוססת–עלות, הנשענת על סטטיסטיקות, או כתוצאה מבאג ב–optimizer. ודא שאתה משתמש ב–hints הקשורים לביצועים בקוד תפעולי רק לאחר שמיצית את כל האמצעים האחרים, כולל שינויים לשאילתה, וידוא שהסטטיסטיקות מעודכנות, ווידוא שלסטטיסטיקות יש אחוז דגימה מספק וכו'.

תוכל למצוא מידע מפורט על hints נתמכים שונים ב– Books Online. אני אשתמש ב–hints במספר מקרים בספרים אלו ואסביר אותם לפי הקשר.

לסיום סעיף זה, ארצה להציג hint חדש ומעניין ב– SQL Server 2005 שאפשר להחשיב כ–hint האולטימטיבי – USE PLAN. Hint זה מאפשר לך לספק ערך XML המחזיק מידע שלם של תוכנית עבודה כדי להכריח את ה–optimizer להשתמש בתוכנית שאתה סיפקת. זכור שבאפשרותך להשתמש באפשרות ה– SHOWPLAN_XML session לייצור תוכנית XML. כדי לראות הדגמה של מה שקורה כאשר אתה משתמש ב–hint זה, ראשית הרץ את הקוד הבא ליצירת תוכנית ה–XML:

```
SET SHOWPLAN_XML ON;
GO
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= 2147483647;
GO
SET SHOWPLAN_XML OFF;
```

כעת הרץ את השאילתה, המספקת את ערך תוכנית ה–XML ב– USE PLAN hint כך:

```
DECLARE @oid AS INT;
SET @oid = 1000000;

SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= @oid
OPTION (USE PLAN
N'<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan" Version="1.0" Build="9.00.1399.06">
 <BatchSequence>
   <Batch>
     <Statements>
       <StmtSimple StatementText="SELECT orderid, custid, empid,
shipperid, orderdate&#xD;&#xA;FROM dbo.Orders&#xD;&#xA;WHERE orderid
&gt;= 2147483647;&#xD;&#xA;" StatementId="1" StatementCompId="1"
StatementType="SELECT" StatementSubTreeCost="0.00657038"
StatementEstRows="1" StatementOptmLevel="FULL" StatementOptmEarlyAbort
Reason="GoodEnoughPlanFound">
         <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true"
CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false" ANSI_PADDING="false"
ANSI_WARNINGS="false" NUMERIC_ROUNDABORT="false" />
         <QueryPlan CachedPlanSize="14">
           <RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner Join"
```

```
EstimateRows="1" EstimateIO="0" EstimateCPU="4.18e-006" AvgRowSize="40"
EstimatedTotalSubtreeCost="0.00657038" Parallel="0" EstimateRebinds="0"
EstimateRewinds="0">
...
            <ParameterList>
              <ColumnReference Column="@1" ParameterCompiledValue="(2147483647)" />
            </ParameterList>
          </QueryPlan>
        </StmtSimple>
      </Statements>
    </Batch>
  </BatchSequence>
</ShowPlanXML>');
```

שים לב שערך ה–XML בקוד לעיל מוצג בצורה מקוצרת. כמובן, עליך לציין את ערך ה–XML המלא. SQL Server 2005 תומך גם במדריך תוכנית (plan guide) חדש, המאפשר לך לחבר תוכנית XML לשאילתה כאשר אינך יכול או אינך רוצה לשנות את מלל השאילתה ישירות על ידי הוספת hints. תוכל להשתמש בפרוצדורה המאוחסנת sp_create_plan_guide ליצירת מדריך תוכנית לשאילתה. תוכל למצוא פרטים נוספים בנושא זה ב– Books Online.

SQL Server 2005 מציג גם מספר hints מעניינים אחרים, ביניהם ה– RECOMPILE hints ו– OPTIMIZE FOR. אדון באלה בספר **תכנות T-SQL** כחלק מהדיון על קומפילציות וקומפילציות–חוזרות של פרוצדורות מאוחסנות.

Traces/Profiler

יכולות ה–trace של SQL Server נותנות לך כלים חזקים ביותר לכוונון ולמטרות אחרות. אחד מהיתרונות הגדולים שיש ל–tracing על פני כלים חיצוניים אחרים הוא שאתה מקבל מידע לגבי אירועים שהתרחשו בתוך השרת ברכיבים שונים. Tracing מאפשר לך לאתר בעיות ביצועים, התנהגות יישום, deadlocks, מידע audit ודברים רבים נוספים. Tracing לאיתור deadlocks יוצג בספר **תכנות T-SQL**. בסוף פרק זה, אכוון אותך למקורות נוספים המכסים tracing ו–profiler.

Database Engine Tuning Advisor

ה– DTA (Database Engine Tuning Advisor) הוא כלי מתקדם שנקרא Index Tuning Wizard ב– SQL Server 2000. DTA ייתן לך המלצות לעיצוב אינדקסים בהתבסס על ניתוח עומס שאתה מספק לו כקלט. קלט העומס יכול להיות קובץ או טבלת trace, והוא יכול להיות גם קובץ קוד המכיל שאילתות T-SQL. יתרון אחד של DTA הוא שהוא משתמש

ב–optimizer של SQL Server כדי לבצע הערכות עלות – אותו optimizer המייצר תוכניות עבודה לשאילתות שלך. בין התכונות החדשות של SQL Server 2005 הזמינות ב–DTA נמצאות המלצות חיצה (partitioning) לטבלאות ואינדקסים. שים לב שאתה יכול להריץ DTA במצב batch על ידי שימוש בתוכנית השרות dta.exe.

כוונון אינדקסים

סעיף זה עוסק בכוונון אינדקסים, שהוא פן חשוב של כוונון שאילתות. אינדקסים הם מבנים המשרתים פעולות חיפוש ומיון. הם מפחיתים את הצורך ב–I/O כאשר מחפשים נתונים ובמיון כאשר אלמנטים מסוימים בתוכנית צריכים או יכולים להפיק תועלת מנתונים ממוינים. בעוד שמספר היבטים של כוונון עשויים לשפר ביצועים באחוז צנוע, כוונון אינדקסים עשוי לעיתים קרובות לשפר ביצועי שאילתה בצורה משמעותית. לפיכך, אם אתה אחראי על כוונון, רצוי שתשקיע זמן בלימוד מעמיק של אינדקסים. כאן אסקור היבטים של כוונון אינדקסים הרלוונטיים לספר זה, ובסוף הפרק אכוון אותך למקורות אחרים לצורך קבלת מידע נוסף.

אתחיל בתיאור מבני טבלאות ואינדקסים הרלוונטיים לדיון שלנו. לאחר מכן אדון בשיטות גישה לאינדקסים המשמשים את ה–optimizer ואסיים את הסעיף בהצגת סרגל לאופטימיזציה של אינדקסים.

מבני טבלאות ואינדקסים

בטרם נצלול לשיטות גישה לאינדקסים, עליך להכיר מבני טבלאות ואינדקסים. סעיף זה יתאר דפים ו–extents, heaps, אינדקס–clustered ואינדקס–nonclustered.

דפים ו–Extents

דף הוא יחידה בת 8-KB בה מאחסן SQL Server נתונים. הוא יכול להכיל נתוני טבלה או אינדקס, נתוני תוכנית עבודה, bitmaps להקצאה, מידע על מקום פנוי, וכו'. דף הוא יחידת ה–I/O הקטנה ביותר ש– SQL Server יכול לקרוא או לכתוב. ב– SQL Server 2000 ובגרסאות קודמות, שורה לא יכלה להתפרש על פני מספר דפים והייתה מוגבלת ל–8060 בתים ברוטו (מלבד נתוני אובייקטים גדולים). המגבלה הייתה בגלל גודל הדף (8192 בתים), שממנו הופחת גודל הכותרת העילית (96 בתים), מצביע לשורה הנשמרת בסוף הדף (2 בתים), ומספר בתים נוספים שהיו שמורים לשימוש עתידי. SQL Server 2005 מציג תכונה חדשה הנקראת row-overflow data, המסירה את המגבלה על גודל שורה עבור טבלאות המכילות VARCHAR, NVARCHAR, VARBINARY, SQL_VARIANT, או טורים מטיפוסי CLR מוגדרי–משתמש. כל טור כזה יכול להגיע ל–8000 בתים, מה שמאפשר לשורה להתפרש על פני מספר עמודים.

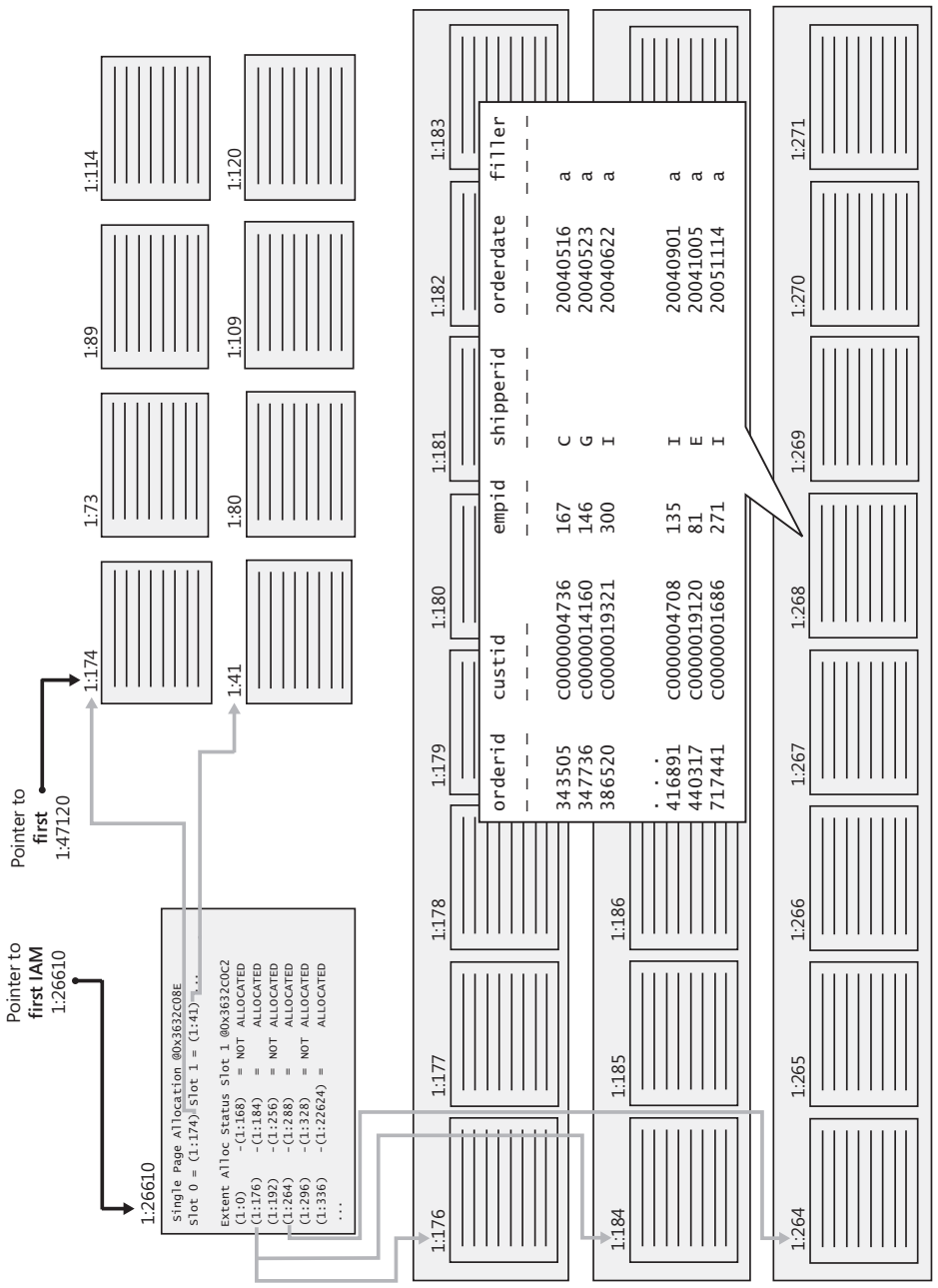
זכור שדף הוא יחידת ה–I/O הקטנה ביותר אותה יכול SQL Server לקרוא או לכתוב. אפילו אם SQL Server צריך לגשת לשורה יחידה, עליו לטעון את כל הדף ל–cache ולקרוא אותה משם. שאילתות המערבות בעיקר מניפולציית נתונים לרוב מוגבלות בעיקר על ידי עלות ה–I/O שלהן. כמובן, קריאה פיסית של דף יקרה הרבה יותר מקריאה לוגית של דף הנמצא כבר ב–cache. קשה להגיע למספר שייצג את יחס הביצועים ביניהם, שכן קיימים מספר גורמים המעורבים בעלות הקריאה, ביניהם סוג שיטת הגישה שהייתה בשימוש, רמת הפרגמנטציה של הנתונים וגורמים נוספים. אם אתה ממש צריך אומדן כלשהו, השתמש ב–1/50 – כלומר, קריאה לוגית תהיה, בהערכה גסה, פי 50 מהירה יותר מאשר קריאה פיסית. אך אני ממליץ בחום לא להסתמך על שום מספר ככלל אצבע.

Extents הם יחידות הקצאה של 8 דפים רציפים. כאשר טבלה או אינדקס זקוקים למקום נוסף לנתונים, SQL Server מקצה extent מלא לאובייקט. ישנו יוצא מן הכלל אחד לגבי אובייקטים קטנים: אם האובייקט קטן מ– 64KB, SQL Server לרוב מקצה דף יחיד כאשר נדרש מקום נוסף, ולא extent מלא. דף זה יכול לשבת בתוך extent מעורב ששמונת דפיו שייכים לאובייקטים שונים. מספר פעילויות של מחיקת נתונים – למשל, מחיקת טבלה וקיצוץ (truncation) טבלה – מפנים extents מלאים. פעילויות כאלה נרשמות בלוג בצורה מינימלית; לפיכך הן מהירות מאוד בהשוואה למשפט DELETE הנרשם בלוג בצורה מלאה. כמו כן, מספר פעילויות קריאה – כגון קריאות read-ahead, המופעלות לרוב עבור סריקת טבלאות או אינדקסים גדולים – יכולות לקרוא נתונים ברמת ה–extent. החלק היקר ביותר של פעולת I/O היא התנועה של זרוע הדיסק, בעוד שפעולת הקריאה או הכתיבה המגנטית בפועל יקרה הרבה פחות; לפיכך, קריאת דף עשויה לארוך כמעט כמו קריאת extent מלא.

Heap

Heap הוא טבלה ללא אינדקס–clustered. המבנה נקרא heap משום שהנתונים אינם מסודרים בשום סדר; אלא הם פרושים כקבוצה של extents. תרשים 3–17 מראה כיצד טבלת Orders שלנו עשויה להיראות כאשר היא מאורגנת כ–heap.

תרשים 3-17: Heap



המבנה היחיד שעוקב אחר הנתונים השייכים ל–heap הוא דף עם מפת סיביות (או סדרת דפים אם נדרש) הנקרא IAM (Index Allocation Map). בדף זה יש מצביעים ל–8 הדפים הראשונים המוקצים מ–extents שונים, וסיבית מייצגת לכל extent בתחום של 4GB בקובץ. הסיבית היא 0 אם ה–extent אותו היא מייצגת אינו שייך לאובייקט שבבעלותו דף ה–IAM, ו–1 אם הוא שייך. אם דף IAM אחד אינו מספיק לכסות את כל נתוני האובייקט, SQL Server יתחזק שרשרת של דפי IAM. SQL Server משתמש בדפי IAM כדי לנוע דרך נתוני האובייקט כאשר האובייקט צריך להיסרק. SQL Server טוען את דף ה–IAM הראשון של האובייקט, ואז מכוון את זרוע הדיסק ברצף להביא את ה–extents בסדר הפיסי שלהם מהדיסק.

כפי שניתן לראות בתרשים, SQL Server מחזיק מצביעים פנימיים לדף ה–IAM הראשון ולדף הנתונים הראשון של ה–heap.

אינדקס–Clustered

כל האינדקסים ב– SQL Server נבנים **כעצים מאוזנים (balanced trees)**. ההגדרה של עץ מאוזן (הובאה מ– www.nist.gov) היא: "עץ בו אין אף עלה הרחוק מהשורש יותר מאשר כל עלה אחר".

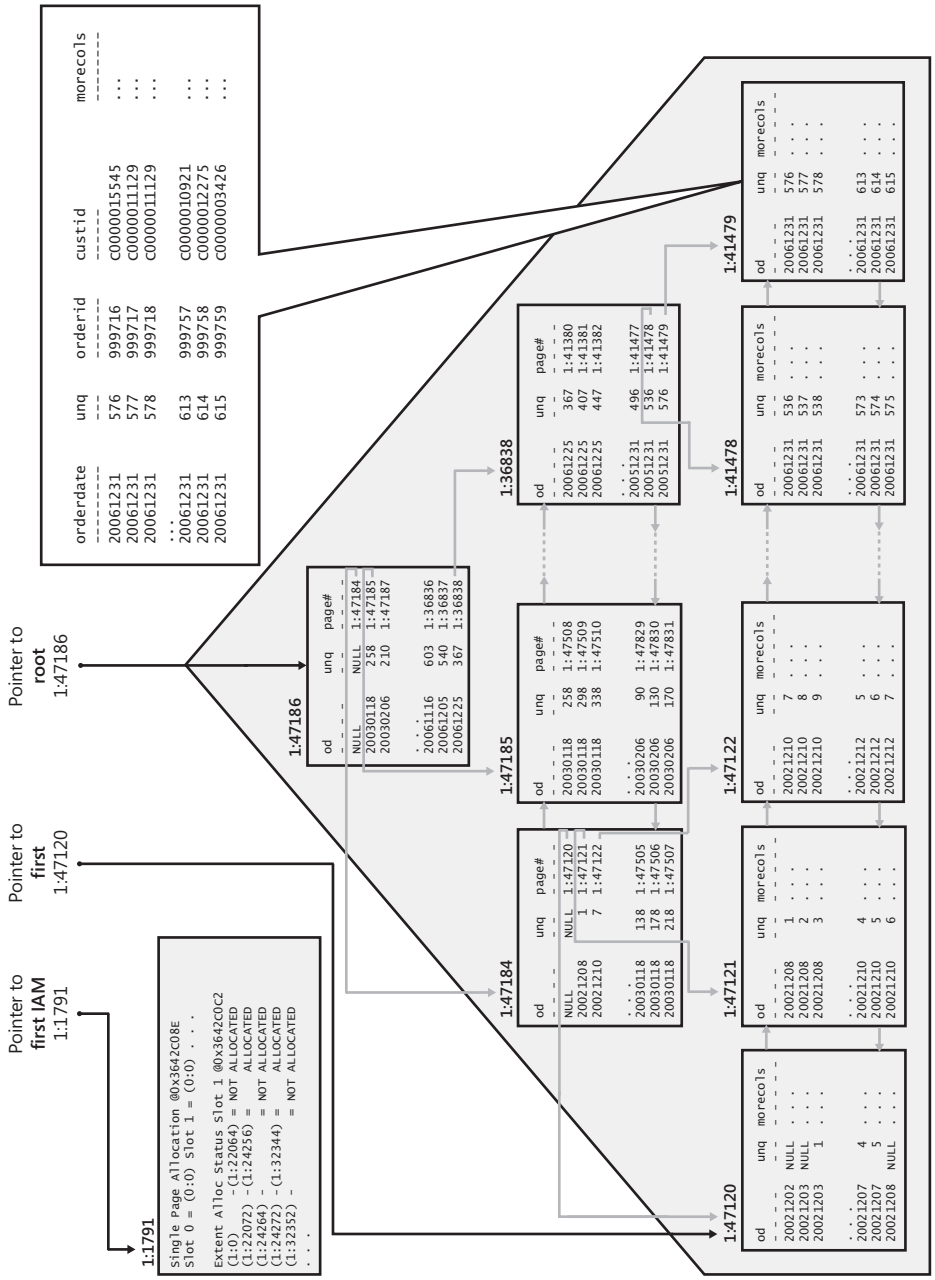
מידע נוסף: אם אתה מעוניין ברקע האלגוריתמי התיאורטי לעצים מאוזנים, אנא פנה לכתובת: http://www.nist.gov/dads/HTML/balancedtree.html, ולספר The Art of Computer Programming, Volume 3: Sorting and Searching (מהדורה שנייה) מאת Donald E. Knuth (Addison-Wesley Professional, 1998).

אינדקס–clustered בנוי כעץ מאוזן, והוא מחזיק את כל נתוני הטבלה ברמת העלה שלו. האינדקס–clustered אינו עותק של הנתונים אלא **הוא** הנתונים. את המבנה של אינדקס– clustered ב– SQL Server אתאר דרך השרטוט המוצג בתרשים 18–3.

התרשים מציג כיצד עשויה טבלת Orders להיראות כאשר היא מאורגנת באינדקס–clustered בו הטור orderdate מוגדר כטור המפתח של האינדקס. לאורך ספרים אלו אתייחס לטבלה בעלת אינדקס–clustered **כטבלה–clustered**. כפי שניתן לראות בתרשים, שורות הנתונים המלאות של טבלת Orders מאוחסנות **ברמת העלה** של האינדקס. שורות הנתונים מאורגנות ברמת העלה בצורה ממוינת בהתבסס על טורי המפתח של האינדקס (orderdate במקרה שלנו). רשימה מקושרת דו–כיוונית מתחזקת את הסדר הלוגי הזה, אך שים לב שבהתאם לרמת הפרגמנטציה של האינדקס, הסדר הפיסי של הדפים על הדיסק עשוי לא להקביל לסדר הלוגי המתוחזק על ידי הרשימה המקושרת.

כמו כן שים לב שעם כל שורת עלה, האינדקס מחזיק ערך הנקרא **uniquifier** (מקוצר ל–unq בדוגמה). ערך זה מונה שורות שלהן אותו ערך מפתח, והוא משמש יחד עם ערך המפתח לזהות ייחודית שורות כאשר טורי המפתח של האינדקס אינם ייחודיים. בהמשך, כאשר נדון באינדקסים–nonclustered, ארחיב את הדיבור על ההיגיון שמאחורי ארכיטקטורה זו והצורך לזהות ייחודית שורה באינדקס–clustered.

תרשים 18-3: אינדקס/טבלה-clustered



יתר הדיון בסעיף זה רלוונטי באותה מידה הן לאינדקסים–clustered והן לאינדקסים–nonclustered, אלא אם כן מצוין בפירוש אחרת. כאשר על SQL Server לבצע פעולות סריקה ממוינות (או סריקה חלקית ממוינת) ברמת העלה של האינדקס, הוא עושה זאת על ידי סריקת הרשימה המקושרת. שים לב שמלבד הרשימה המקושרת, SQL Server מתחזק גם דף (או דפי) IAM כדי למפות את הנתונים המאוחסנים באינדקס לפי סדר פיסי על הדיסק. SQL Server ישתמש לרוב בדף IAM כאשר עליו לבצע סריקות לא–ממוינות של רמת העלה של האינדקס. הבדל הביצועים בין סריקות ממוינות ולא ממוינות של האינדקס יהיה תלוי ברמת הפרגמנטציה באינדקס. זכור שהחלק היקר ביותר של פעולת I/O הוא התנועה של זרוע הדיסק. סריקה ממוינת באינדקס ללא פרגמנטציה בכלל תהיה דומה בביצועיה לסריקה לא–ממוינת, בעוד שהראשונה תהיה איטית משמעותית באינדקס בעל רמה גבוהה של פרגמנטציה.

מעל רמת העלה של האינדקס, האינדקס מתחזק רמות נוספות, כל אחת ממפה את הרמה שמתחתיה. כל שורה בדף אינדקס שאינו ברמת עלה מצביעה לדף שלם ברמה שמתחתיה. השורה מכילה שני אלמנטים: ערך טור המפתח של השורה הראשונה בדף האינדקס עליו היא מצביעה, ומצביע בן 6 בתים המצביע לדף זה. המצביע מחזיק את מספר הקובץ במסד הנתונים ואת מספר הדף בקובץ. כאשר SQL Server בונה אינדקס, הוא מתחיל מרמת העלה ומוסיף רמות מעליה. הוא מפסיק ברגע שרמה מכילה דף בודד, הידוע גם כדף **שורש (root)**.

SQL Server מתחיל תמיד בדף השורש כאשר עליו לנווט למפתח מסוים בעלה, תוך שהוא משתמש בשיטת גישה הנקראת **index seek**, עליה ארחיב בהמשך הפרק. פעולת הסריקה "תקפוץ" מהשורש לדף הרלוונטי ברמה הבאה, והיא תמשיך לקפוץ מרמה אחת לבאה עד שתגיע לדף המכיל את המפתח המבוקש בעלה. זכור שכל דפי העלה נמצאים באותו מרחק מהשורש, והמשמעות היא שפעולת סריקה תעלה מספר קריאות דף כמספר הרמות באינדקס. דפוס ה–I/O של קריאות אלה הוא I/O **רנדומאלי**, בניגוד ל–I/O רציף, משום שבצורה טבעית הדפים הנקראים על ידי פעולת סריקה ישבו לעיתים רחוקות בסמיכות זה לזה.

במונחי הערכות הביצועים שלנו, חיוני לדעת מה מספר הרמות באינדקס משום שמספר זה יהיה העלות של פעולת סריקה במונחי קריאות דף, וישנן תוכניות עבודה המפעילות מספר פעולות סריקה שוב ושוב (למשל, אופרטור Nested Loops – join). עבור אינדקס קיים, אתה יכול לקבל את המספר הזה על ידי קריאה לפונקציה INDEXPROPERTY עם המאפיין IndexDepth. אך עבור אינדקס שלא ייצרת עדיין, עליך להכיר את החישובים שיאפשרו לך להעריך את מספר הרמות שיכיל האינדקס.

האופרנדים והצעדים הנדרשים לחישוב מספר הרמות באינדקס (קרא לו L) הם כלהלן (זכור שחישובים אלו נכונים לאינדקסים–clustered ו–nonclustered, אלא אם כן מצוין בפירוש אחרת):

- **מספר השורות בטבלה (קרא לו num_rows):** במקרה שלנו הוא 1,000,000.

- **גודל ממוצע ברוטו של שורת עלה (קרא לו leaf_row_size):** באינדקס–clustered, זהו למעשה גודל שורת הנתונים. ב"ברוטו" אני מתכוון שעליך לקחת את התקורה הפנימית של שורה ואת המצביע בן 2 הבתים – המאוחסן בסוף העמוד – המצביע לשורה. תקורת השורה מערבת לרוב מספר בתים. בטבלת Orders שלנו, הגודל הממוצע ברוטו של שורת נתונים הוא בערך 200 בתים.
- **צפיפות ממוצעת של דף עלה (קרא לה page_density):** ערך זה הוא אחוז המילוי הממוצע של דפי עלה. סיבות לכך שדפים אינם מלאים לחלוטין כוללות: מחיקת נתונים, פיצולי דפים הנגרמים בגלל הכנסת שורות לעמודים מלאים לחלוטין, קיומן של שורות גדולות מאוד, ובקשות מפורשות לא למלא דפים לחלוטין בנתונים על ידי ציון הערך fillfactor כאשר בונים את האינדקסים מחדש. במקרה שלנו, יצרנו אינדקס–clustered על טבלת Orders לאחר שמילאנו אותה בנתונים, לא הוספנו שורות לאחר יצירת האינדקס–clustered, ולא ציינו ערך fillfactor. לפיכך, page_density במקרה שלנו קרוב ל–100 אחוז.
- **מספר השורות הנכנסות בדף עלה (קרא לו rows_per_leaf_page):** הנוסחה לחישוב ערך זה היא: (page_size - header_size) * page_density / leaf_row_size. שים לב שאם יש לך הערכה טובה של page_density, אין צורך לעגל מטה ערך זה, שכן העובדה ששורה אינה יכולה להתפרש על פני מספר דפים (עם יוצאי הדופן שהוזכרו לעיל) נלקחת כבר בחשבון בערך ה–page_density. במקרה כזה, תרצה להשתמש במספר שהתקבל כפי שהוא אפילו אם אינו שלם. מצד שני, אם אתה רק מעריך ש–page_density יהיה קרוב ל–100 אחוז, כמו במקרה שלנו, התעלם מהאופרנד page_density בחישוב ועגל את התוצאה למטה. במקרה שלנו, rows_per_leaf_page שווה ל– floor((8192 – 96) / 200) = 40.
- **מספר הדפים המוחזקים בעלה (קרא לו num_leaf_pages):** זוהי נוסחה פשוטה – num_rows / rows_per_leaf_page. במקרה שלנו, המספר הוא:

1,000,000 / 40 = 25,000

- **גודל ברוטו ממוצע של שורה שאינה ברמת העלה (קרא לו non_leaf_row_size):** שורה שאינה ברמת העלה מכילה את טורי המפתח של האינדקס (במקרה שלנו, orderdate בלבד, שהוא בן 8 בתים); ה–uniqifier בן 4 הבתים (הקיים רק באינדקס–clustered שאינו ייחודי); מצביע הדף, שהוא בן 6 בתים; מספר בתים נוספים של תקורה פנימית, המגיע ל–5 בתים במקרה שלנו; והמצביע row offset בסוף הדף, שהוא בן 2 בתים. במקרה שלנו, הגודל ברוטו של שורה שאינה ברמת העלה הוא 25 בתים.
- **מספר השורות שיכולות להיכנס בדף שאינו ברמת העלה (קרא לו rows_per_non_leaf_page):** הנוסחה לחישוב ערך זה דומה לחישוב rows_per_leaf_page. לשם הפשטות, אתעלם מגורם צפיפות דף שאינו ברמת עלה, ואחשב את הערך בצורה הבאה: floor((page_size - header_size) / non_leaf_row_size), שבמקרה שלנו מגיע ל– floor((8192 – 96) / 25) = 323.

- **מספר הרמות מעל העלה (קרא לו L–1):** ערך זה מחושב על ידי הנוסחה הבאה: $\text{ceiling}(\log_{\textit{rows per non leaf page}}(\text{num\_leaf\_pages}))$. במקרה שלנו, L-1 שווה ל– $\text{ceiling}(\log_{323}(25000)) = 2$. כמובן, אתה פשוט צריך להוסיף 1 כדי לקבל L, אשר במקרה שלנו הוא 3.

תרגיל זה מוביל אותי לנקודה חשובה ביותר עליה אשען בדיונים שלי על ביצועים. תוכל לשחק עם הנוסחה ולראות שעד מספר אלפים של שורות, לאינדקס שלנו יהיו שתי רמות. שלוש רמות יכסו עד בערך 4,000,000 שורות, ו–4 רמות יכסו עד 4,000,000,000 שורות. עם אינדקסים–nonclustered, הנוסחאות זהות, ההבדל הוא רק בזה שאתה יכול להכניס יותר שורות בכל דף עלה, כפי שאסביר בהמשך. כך שעם אינדקסים–nonclustered הגבול העליון לכל מספר רמות מכסה אפילו יותר שורות בטבלה. הנקודה היא שבטבלת ה–Orders שלנו, לכל האינדקסים יש 3 רמות. זכור שעלות פעולת seek מול אינדקס במונחי מספר קריאות דפים, היא כמספר הרמות באינדקס. ובאופן כללי, עם טבלאות קטנות רוב האינדקסים יהיו באופן טיפוסי בעלי עד 2 רמות, ועם טבלאות גדולות, הם יהיו באופן טיפוסי בעלי 3 או 4 רמות, אלא אם כן הגודל הכולל של מפתחות האינדקס הוא גדול. זכור את המספרים הללו לדיונים שלנו בהמשך.

אינדקס–Nonclustered על Heap

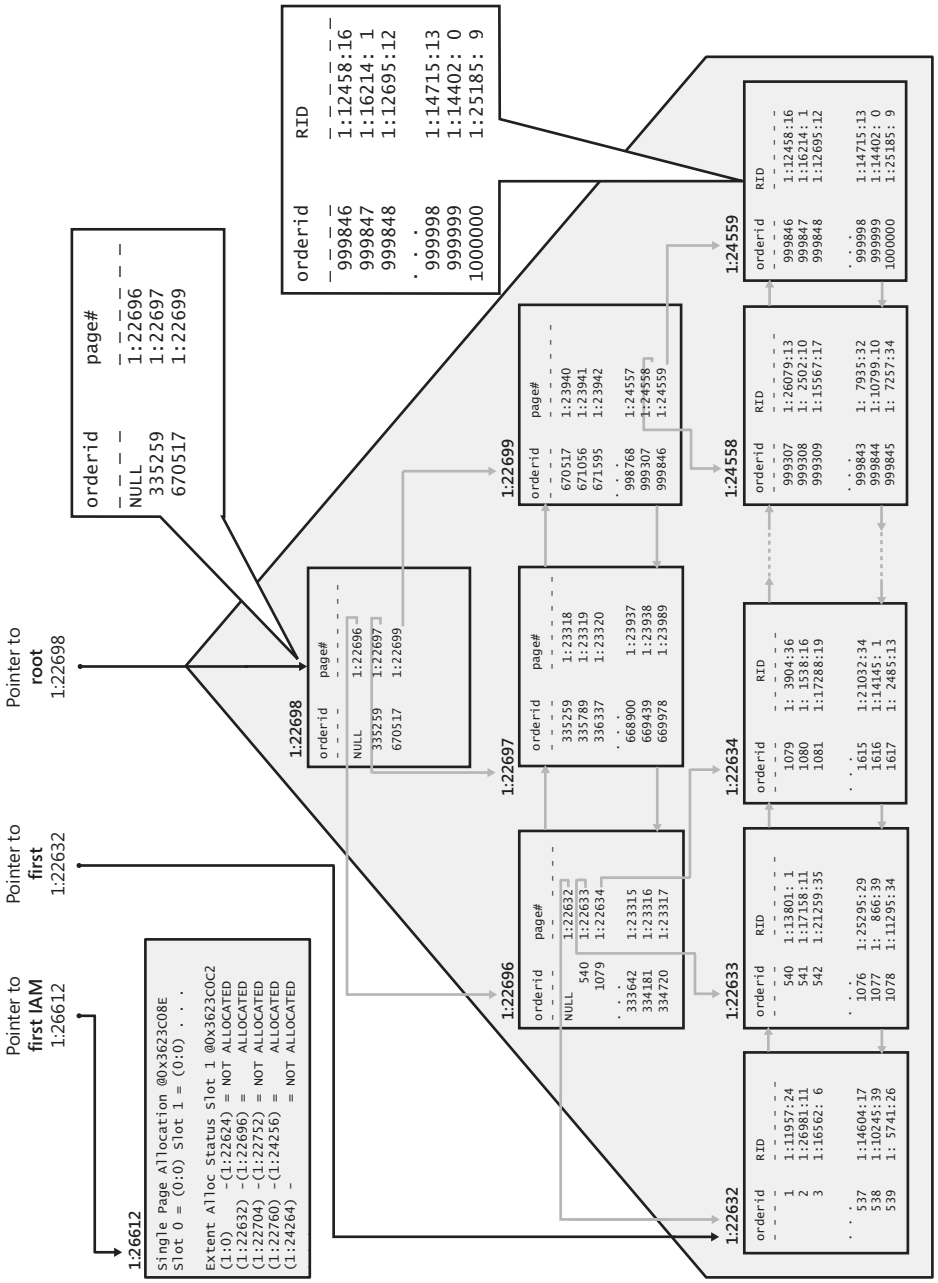
אינדקס–nonclustered גם הוא מעוצב כעץ מאוזן, ומבחינות רבות הוא דומה לאינדקס–clustered. ההבדל היחיד הוא ששורת עלה באינדקס–nonclustered מכילה רק את טורי מפתחות האינדקס וערך מאתר שורה (row locator) המצביע לשורת נתונים מסוימת. תוכן מאתר השורה תלוי בשאלה האם הטבלה היא heap או טבלה–clustered. סעיף זה מתאר אינדקסים–nonclustered על heap, והסעיף הבא יתאר אינדקסים–nonclustered על טבלה–clustered.

תרשים 3–19 מתאר את האינדקס–nonclustered שנוצר על ידי אילוץ המפתח הראשי שלנו (PK_Orders) המגדיר את הטור orderid כטור המפתח.

שורת עלה באינדקס–nonclustered משתמשת במאתר שורה כדי להצביע לשורת נתונים. מאתר שורה הוא מצביע פיסי בן 8 בתים הנקרא RID. הוא מורכב ממספר הקובץ במסד הנתונים, מספר דף היעד בקובץ, ומספר השורה בדף היעד (מבוסס–אפס). כאשר מחפשים אחר שורת נתונים מסוימת דרך האינדקס, SQL Server יהיה חייב לעקוב אחר פעולת הסריקה עם פעולת RID lookup, שהיא קריאת הדף המכיל את שורת הנתונים. לפיכך, העלות של RID lookup היא קריאת דף אחד. עבור lookup אחד או מספר קטן מאוד של lookups העלות אינה גבוהה, אך עבור מספר גדול של lookups העלות עשויה להיות גבוהה מאוד משום ש– SQL Server בסופו של דבר קורא דף שלם לכל שורה אותה מחפשים. עבור שאילתות תחום המשתמשות באינדקס–nonclustered, וסדרת lookups – אחד לכל מפתח מתאים – העלות המצטברת של פעולות ה–lookup לרוב מהווה את מרבית עלות השאילתה. אדגים נקודה זו בסעיף "שיטות גישה לאינדקסים". באשר לעלות של פעולת סריקה (scan), זכור שהנוסחאות שסיפקתי קודם לכן רלוונטיות באותה מידה

לאינדקסים–nonclustered. ההבדל היחיד הוא שה–leaf_row_size קטן יותר, ולפיכך rows_per_leaf_page יהיה גבוה יותר. אך הנוסחאות זהות.

תרשים 19–3: אינדקס–nonclustered על heap



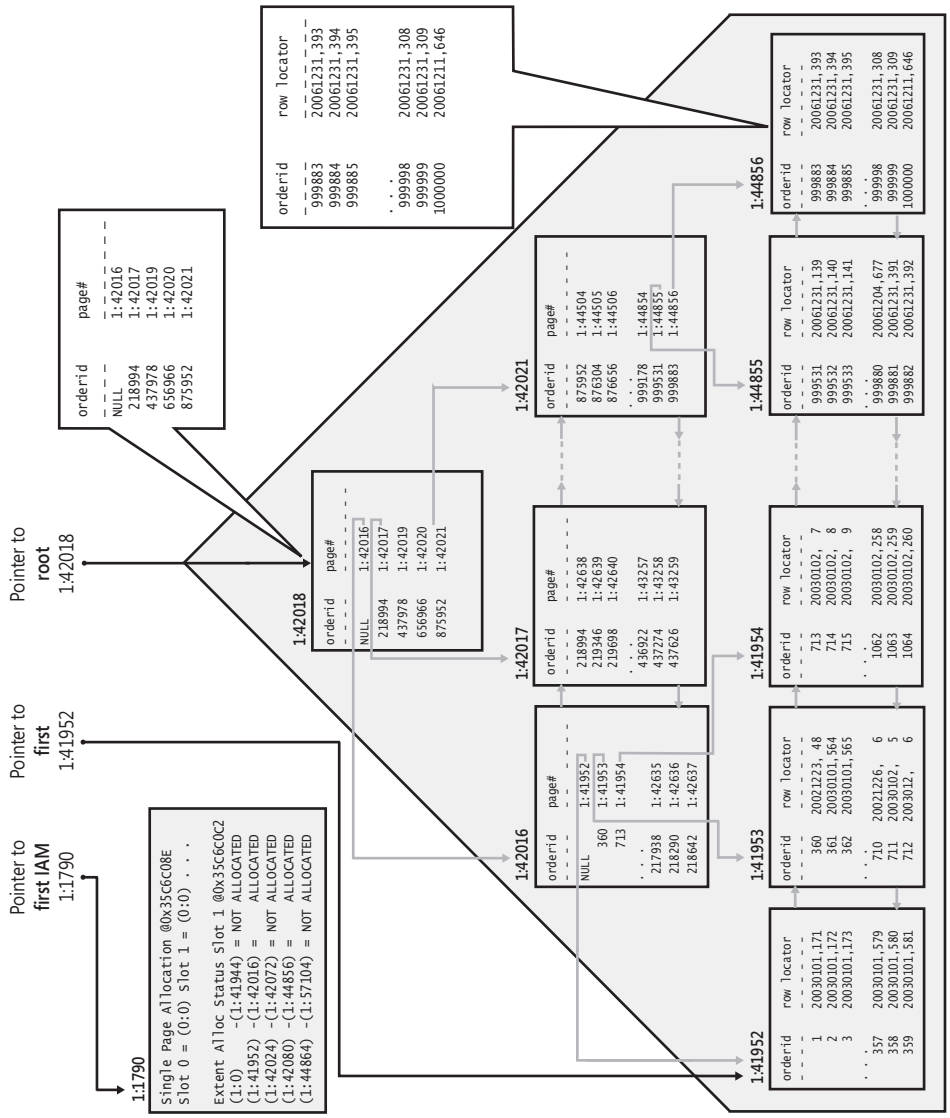
אינדקס–Nonclustered על טבלה–Clustered

החל בגרסת SQL Server 7.0, לאינדקסים–nonclustered שנוצרו על טבלה–clustered יש ארכיטקטורה שונה מאילו על heap. ההבדל היחיד הוא שמאתר השורה באינדקס–nonclustered שנוצר על טבלה–clustered הוא ערך הנקרא מפתח clustering (clustering key), בניגוד ל–RID. מפתח ה–clustering מורכב מהערכים של מפתחות האינדקס–clustered מהשורה אליה מצביעים, וה–uniquifier (אם קיים). הרעיון הוא להצביע לשורה בצורה "לוגית" בניגוד ל"פיסית". ארכיטקטורה זו עוצבה בעיקר למערכות OLTP, בהן אינדקסים–clustered סובלים לעיתים קרובות מפיצולי דפים רבים כאשר מוכנסים נתונים. בפיצול דף, מחצית השורות מהדף המפוצל מועברות פיסית לדף חדש שהוקצה. אם אינדקסים–nonclustered היו שומרים מצביעים פיסיים לשורות, כל המצביעים לשורות הנתונים שהוזזו היו צריכים לעבור שינוי כדי לשקף את המיקומים הפיסיים החדשים שלהם – ודבר זה נכון לכל המצביעים הרלוונטיים בכל האינדקסים–nonclustered. במקום זאת, SQL Server שומר מצביעים לוגיים שאינם משתנים כאשר שורות נתונים זזות פיסית.

תרשים 3–20 מתאר כיצד עשוי להיראות האינדקס–nonclustered – PK_Orders; האינדקס מוגדר עם orderid כטור המפתח, ולטבלת Orders יש אינדקס–clustered המוגדר עם orderdate כטור המפתח.

פעולת seek המחפשת אחר מפתח מסוים באינדקס–nonclustered (ערך orderid כלשהו) תגיע בסופו של דבר לשורת העלה הרלוונטית ותהיה לה גישה למאתר השורה. מאתר השורה במקרה זה הוא מפתח ה–clustering של השורה עליה מצביעים. כדי לאתר בפועל את השורה עליה מצביעים, פעולת lookup תצטרך לבצע חיפוש שלם בתוך האינדקס–clustered בהתבסס על מפתח ה–clustering שהושג. אדגים שיטת גישה זו בהמשך הפרק. העלות של כל פעולת חיפוש כאן (במונחים של מספר קריאות דף) גובהה כמספר הרמות באינדקס–clustered (3 במקרה שלנו). זאת בהשוואה לקריאת דף יחידה עבור RID lookup כאשר הטבלה היא heap. כמובן, עם שאילתות תחום המשתמשות באינדקס–nonclustered ובסדרת lookups, היחס בין מספר הקריאות הלוגיות במקרה של heap לבין מקרה של טבלה–clustered יהיה קרוב ל–1:L, כאשר L הוא מספר הרמות באינדקס–clustered. בטרם תגרום לך נקודה זו לרוץ ולהסיר את כל האינדקסים–clustered מהטבלאות שלך, זכור שעם כל ה–lookups שעוברים באינדקס–clustered, רמות האינדקס–clustered שאינן–עלה ישבו לרוב ב–cache. לרוב, מרבית הקריאות הפיסיות באינדקס–clustered יהיו מול רמת עלה. לפיכך, העלות הנוספת של lookups מול טבלה–clustered בהשוואה ל–heap מהווה לרוב חלק קטן מסך עלות השאילתה. כעת, לאחר שכיסינו את מידע הרקע לגבי מבני טבלה ואינדקס, הסעיף הבא יתאר שיטות גישה לאינדקסים.

תרשים 20–3: אינדקס–nonclustered על טבלה–clustered



שיטות גישה לאינדקסים

סעיף זה מספק תיאור טכני של השיטות השונות לגישה לאינדקסים; הוא מעוצב לשמש כמקור מידע לדיונים בספרים אלו המערבים ניתוח תוכניות עבודה. בהמשך הפרק, אתאר סרגל אופטימיזציה של אינדקסים המדגים כיצד תוכל לממש ידע זה בצורה מעשית.

אם ברצונך לעקוב אחר הדוגמאות בסעיף זה, הרץ שוב את הקוד בקטע–קוד 1–3 כדי ליצור מחדש את הטבלאות לדוגמה במסד הנתונים Performance בצירוף כל האינדקסים. אדון בכמה שיטות גישה לשימוש מול טבלת Orders הבנויה כ–heap וכמה כשהיא בנויה כטבלה–clustered. לפיכך, אני מציע שתריץ את הקוד בקטע קוד 1–3 גם מול מסד נתונים אחר (נאמר Performance2), לאחר שתשנה את שם מסד הנתונים בקוד בהתאם, ותסיר את המשפט היוצר את האינדקס–clustered על Orders. כאשר אדון בשיטת גישה המערבת טבלה–clustered, הרץ את הקוד מול מסד הנתונים Performance. כאשר אדון ב–heaps, הרץ אותו מול Performance2. זכור גם שקטע קוד 1–3 משתמש ברנדומיזציה כדי למלא נתוני קודי לקוח, קודי עובד, קודי מוביל ותאריכי הזמנה בטבלת Orders. המשמעות היא שהתוצאות שתקבל יהיו מעט שונות משלי.

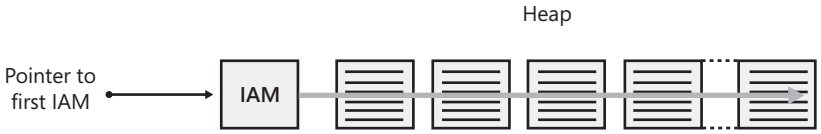
Table Scan/Unordered Clustered Index Scan

table scan או unordered clustered index scan הן פשוט סריקה רציפה של כל דפי הנתונים השייכים לטבלה. השאילתה הבאה מול טבלת Orders הבנויה כ–heap תזדקק ל– table scan:

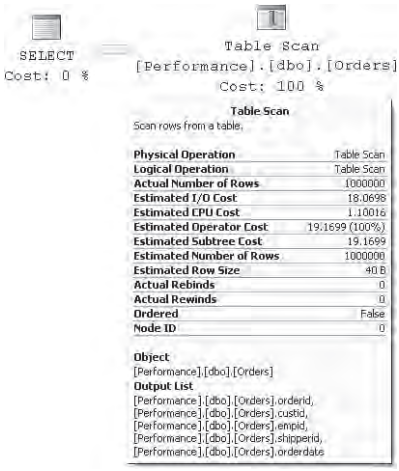
```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders;
```

תרשים 3–21 מציג שרטוט של שיטת גישה זו, ותרשים 3–22 מציג את תוכנית העבודה הגרפית שתקבל עבור שאילתה זו.

תרשים 3–21:
Table scan



תרשים 3–22:
Table scan (תוכנית עבודה)



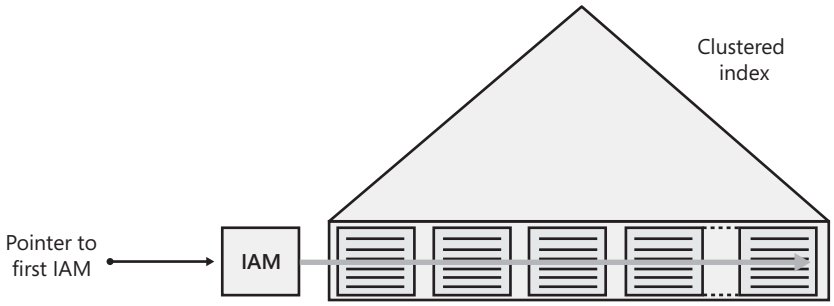
SQL Server ישתמש בדפי ה–IAM של הטבלה כדי לכוון את זרוע הדיסק לסריקת ה–extents השייכים לטבלה ברצף לפי הסדר הפיסי שלהם על הדיסק. מספר הקריאות הלוגיות צריך להיות דומה למספר הדפים שהטבלה מכילה (בסביבות 25,000 במקרה שלנו). שים לב שבסריקות כאלה SQL Server משתמש לרוב באסטרטגיית read-ahead יעילה מאוד שיכולה לקרוא את הנתונים במקטעים גדולים מ–8 KB. כאשר הרצתי שאילתה זו על המערכת שלי, קיבלתי את מדידות הביצועים הבאות מ– STATISTICS IO, STATISTICS TIME ותוכנית העבודה:

- קריאות לוגיות: 24391
- קריאות פיסיות: 0
- קריאות read-ahead: 24408
- זמן מעבד: 971 ms
- זמן שעון: 20265 ms
- עלות תת–עץ משוערת: 19.1699

כמובן, זמני הריצה שקיבלתי אינם אינדיקציה לזמני הריצה שתקבל במערכת תפעולית ממוצעת. אך רציתי להראות אותם לצרכי הדגמה והשוואה.

אם לטבלה יש אינדקס–clustered, שיטת הגישה שתופעל תהיה unordered clustered index scan, כפי שמתואר בתרשים 3–23. היא תפיק את תוכנית העבודה המוצגת בתרשים 3–24.

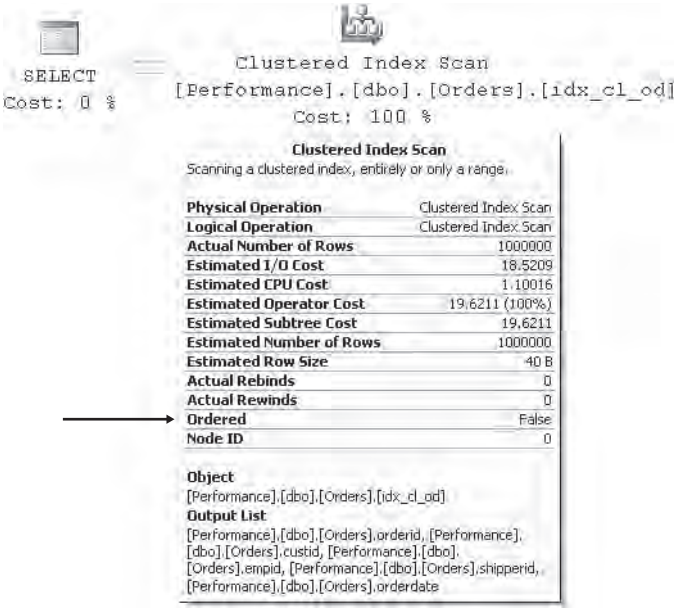
תרשים 3–23: Unordered clustered index scan



שים לב שלמרות שתוכנית העבודה מראה clustered index scan, הפעילות אינה שונה מ– table scan, ולאורך הספר אתייחס אליה לעיתים קרובות פשוט כ– table scan. כפי שנראה בתרשים, כאן SQL Server ישתמש גם בדפי ה–IAM של האינדקס כדי לסרוק את הנתונים ברצף. תיבת המידע של האופרטור Clustered Index Scan אומרת לך שהסריקה לא הייתה לפי הסדר, כלומר ששיטת הגישה לא נשענה על הרשימה המקושרת השומרת את הסדר הלוגי של האינדקס. כמובן, קיבלת סריקה של הדפים לפי הסדר הפיסי שלהם על הדיסק. להלן מדידות הביצועים שקיבלתי משאילתה זו:

- קריאות לוגיות: 25080
- קריאות פיסיות: 1
- קריאות read-ahead: 25071
- זמן מעבד: 941 ms
- זמן שעון: 22122 ms
- עלות תת–עץ משוערת: 19.6211

תרשים 3–24: Unordered clustered index scan (תוכנית עבודה)



Unordered Covering Nonclustered Index Scan

Unordered covering nonclustered index scan דומה בתפיסה ל– unordered clustered index scan. המושג של **אינדקס–מכסה (covering index)** משמעותו שאינדקס–nonclustered מכיל את כל הטורים שהוגדרו בשאילתה. במילים אחרות, אינדקס–מכסה אינו אינדקס בעל תכונות מיוחדות; אלא, הוא הופך לאינדקס–מכסה בעבור שאילתה מסוימת. SQL Server יכול למצוא את כל הנתונים שהוא צריך כדי לענות על השאילתה על ידי גישה לנתוני האינדקס בלבד, ללא הצורך לגשת לשורות הנתונים המלאות. מעבר לכך, שיטת הגישה זהה ל– unordered clustered index scan; אלא שכמובן, רמת העלה של האינדקס–המכסה-nonclustered מכילה פחות דפים מאשר רמת העלה של האינדקס–clustered, משום שגודל

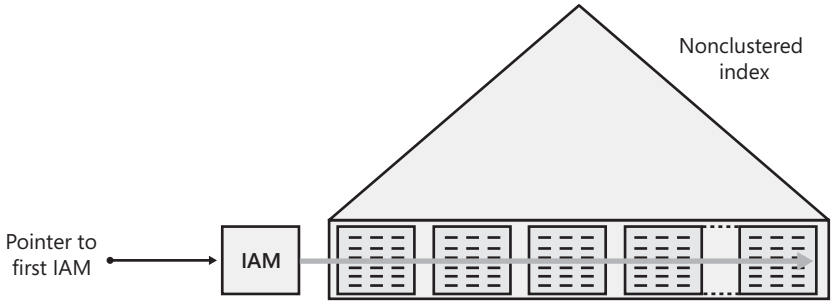
השורה קטן יותר ויותר שורות נכנסות לכל דף. קודם לכן הסברתי כיצד לחשב את מספר הדפים ברמת העלה של אינדקס (clustered או nonclustered).

כדוגמה לשיטת גישה זו, השאילתה הבאה מבקשת את כל ערכי orderid מטבלת Orders:

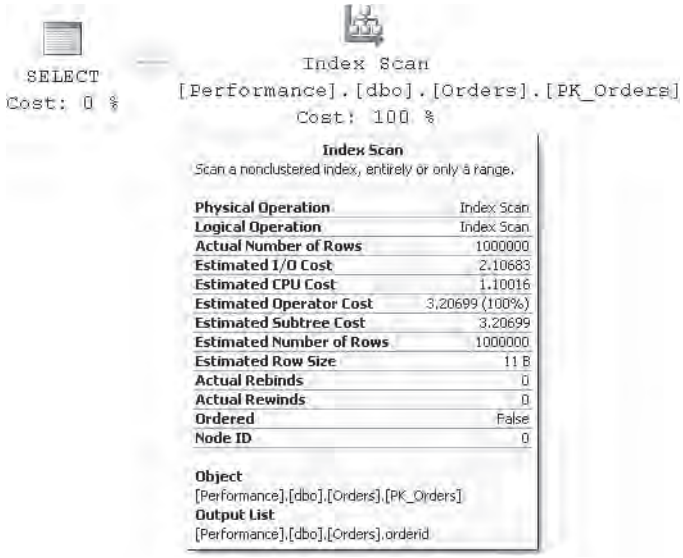
```
SELECT orderid
FROM dbo.Orders;
```

לטבלת Orders שלנו יש אינדקס–nonclustered על הטור orderid (PK_Orders), כשהמשמעות היא שכל קודי ההזמנה של הטבלה ממוקמים ברמת העלה של האינדקס. האינדקס מכסה את השאילתה שלנו. תרשים 3–25 מדגים את שיטת הגישה הזו, ותרשים 3–26 מציג את תוכנית העבודה הגרפית שתקבל עבור שאילתה זו.

תרשים 3–25: Unordered covering nonclustered index scan



תרשים 3–26: Unordered covering nonclustered index scan (תוכנית עבודה)



רמת העלה של האינדקס PK_Orders מכילה פחות מ–3000 דפים, בהשוואה ל–25,000 דפי הנתונים בטבלה. להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 2848
- קריאות פיסיות: 1
- קריאות read-ahead: 2841
- זמן מעבד: 340 ms
- זמן שעון: 12071 ms
- עלות תת–עץ משוערת: 3.20699

Ordered Clustered Index Scan

Ordered clustered index scan היא סריקה מלאה של רמת העלה של האינדקס clustered–לפי הרשימה המקושרת. למשל, השאילתה הבאה, המבקשת את כל ההזמנות ממוינות לפי orderdate, תקבל שיטת גישה כזו בתוכנית שלה:

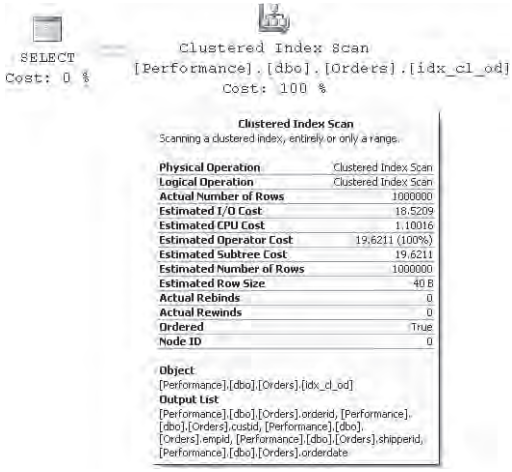
```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
ORDER BY orderdate;
```

תוכל למצוא שרטוט של שיטת גישה זו בתרשים 3–27, ואת תוכנית העבודה של שאילתה זו בתרשים 3–28.

תרשים 3–27: Ordered clustered index scan

שים לב שלא כמו unordered scan של האינדקס, הביצועים של ordered scan יהיו תלויים ברמת הפרגמנטציה של האינדקס – כלומר, אחוז הדפים שאינם–לפי–הסדר ברמת העלה של האינדקס ביחס למספר הכולל של הדפים. דף שאינו–לפי–הסדר הוא דף המופיע לוגית לאחר דף מסוים לפי הרשימה המקושרת, אך פיסית נמצא לפניו. ללא פרגמנטציה כלל, הביצועים של ordered scan של אינדקס צריכים להיות קרובים מאוד לביצועים של unordered scan, משום ששתיהן לבסוף יקראו את הנתונים פיסית בצורה רציפה.

תרשים 3–28: Ordered clustered index scan (תוכנית עבודה)



עם זאת, ככל שרמת הפרגמנטציה גדלה, הבדל הביצועים יהיה משמעותי יותר, לטובת ה– unordered scan כמובן. המסקנות המתבקשות הן שאינך צריך לבקש את הנתונים ממוינים אם אינך זקוק להם ממוינים, ושעליך לטפל בסוגיות פרגמנטציה באינדקסים היוצרים ordered scans גדולות. מאוחר יותר ארחיב את הדיבור על פרגמנטציה והטיפול בה. להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 25080
- קריאות פיסיות: 1
- קריאות read-ahead: 25071
- זמן מעבד: 1191 ms
- זמן שעון: 22263 ms
- עלות תת–עץ משוערת: 19.6211

שים לב שה–optimizer אינו מוגבל לפעילויות סריקה ממוינות בכיוון קדימה. זכור שהרשימה המקושרת היא רשימה מקושרת דו–כיוונית, בה כל דף מכיל מצביעים לדף הבא והקודם. אם היית מבקש סדר מיון יורד, עדיין היית מקבל ordered index scan, אלא שממוינת אחורה (מהזנב לראש) במקום ממוינת קדימה (מהראש לזנב). SQL Server תומך גם באינדקסים יורדים מאז גרסת 2000, אך אלו אינם נדרשים במקרים פשוטים כגון השגת מיונים יורדים. לאינדקסים יורדים יש ערך כאשר אתה יוצר אינדקס על טורי מפתח מרובים בעלי כיוונים מנוגדים בדרישות המיון שלהם – למשל, מיון לפי col1, col2 DESC.

Ordered Covering Nonclustered Index Scan

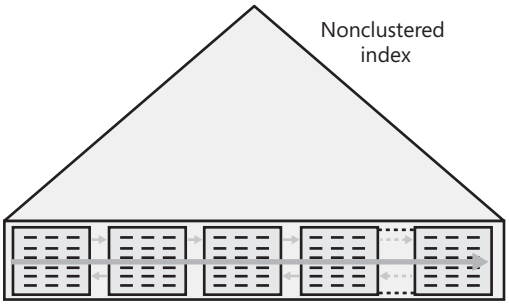
ordered covering nonclustered index scan דומה בתפיסה ל– ordered clustered index scan, כאשר הראשונה מבצעת את שיטת הגישה באינדקס–nonclustered. העלות היא כמובן נמוכה יותר מאשר זו של clustered index scan משום שפחות דפים מעורבים. למשל, האינדקס PK_Orders על הטבלה–clustered שלנו Orders, מכסה את השאילתה הבאה, למרות שאולי לא נראה כך במבט ראשון:

```
SELECT orderid, orderdate
FROM dbo.Orders
ORDER BY orderid;
```

זכור שבטבלה–clustered, אינדקסים–nonclustered ישתמשו במפתחות clustering כמאתרי שורה. במקרה שלנו, המפתחות clustering מכילים את ערכי orderdate, שיכולים לשמש גם למטרות כיסוי. כמו כן, טור המפתח הראשון (ובמקרה שלנו, היחיד) באינדקס–nonclustered הוא טור orderid, המצוין בפסוקית ORDER BY של השאילתה; לפיכך, ordered index scan היא שיטת גישה בה טבעי ל–optimizer לבחור.

תרשים 3–29 מדגים שיטת גישה זו, ותרשים 3–30 מראה את תוכנית העבודה של השאילתה.

תרשים 3–29: Ordered covering nonclustered index scan



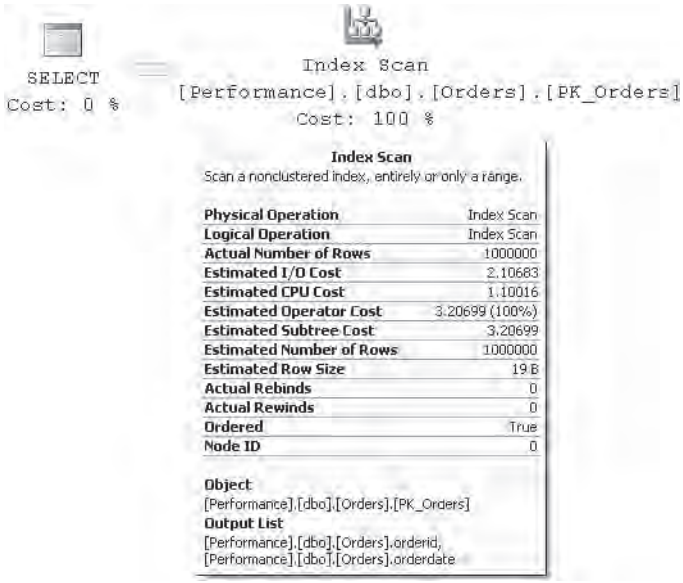
שים לב בתוכנית שהמדד Ordered עבור האופרטור Index Scan בתיבת המידע הצהובה מראה True.

להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 2848
- קריאות פיסיות: 2
- קריאות read-ahead: 2841
- זמן מעבד: 370 ms

- **זמן שעון:** 13582 ms
- **עלות תת–עץ משוערת:** 3.20699

תרשים 3–30: Ordered covering nonclustered index scan (תוכנית עבודה 1)

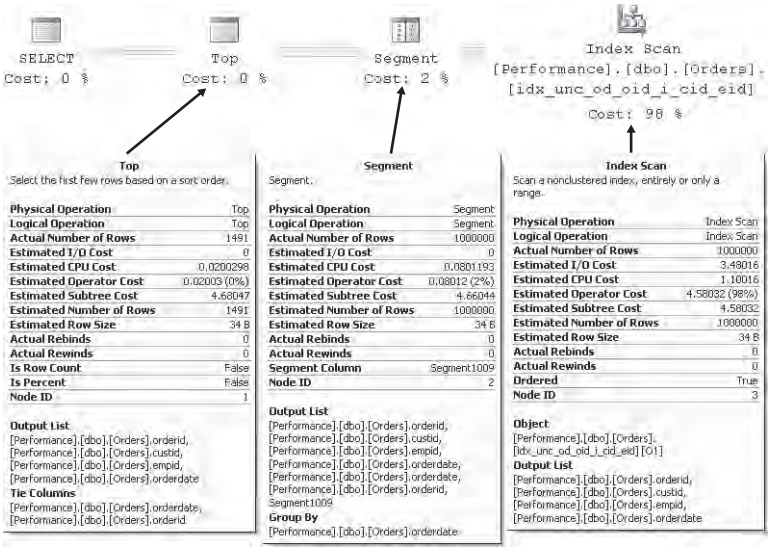


Ordered index scan אינה בשימוש רק כאשר אתה מבקש מפורשות את הנתונים ממוינים; היא בשימוש גם כאשר התוכנית משתמשת באופרטור שעשוי להפיק תועלת מנתוני קלט ממוינים. למשל, בחן את תוכנית העבודה המוצגת בתרשים 3–31 עבור השאילתה הבאה:

```
SELECT orderid, custid, empid, orderdate
FROM dbo.Orders AS O1
WHERE orderid =
  (SELECT MAX(orderid)
   FROM dbo.Orders AS O2
   WHERE O2.orderdate = O1.orderdate);
```

האופרטור Segment מסדר את הנתונים בקבוצות ומעביר קבוצה בכל פעם לאופרטור הבא (TOP במקרה שלנו). השאילתה שלנו מבקשת את ההזמנות עם ה–orderid הגבוה ביותר לכל orderdate. למזלנו, יש לנו אינדקס–מכסה לצורך המשימה (idx_unc_od_oid_i_cid_eid), כאשר טור המפתח הוא (orderdate, orderid), והטורים שאינם–מפתח הכלולים הם (custid, empid). בהמשך הפרק ארחיב את הדיבור על טורים כלולים שאינם–מפתח. הנקודה החשובה לדיון שלנו היא שהאופרטור Segment מסדר את הנתונים בקבוצות לפי ערכי orderdate ומעביר את הנתונים, קבוצה בכל פעם, כאשר השורה האחרונה בכל קבוצה היא ה–orderid הגדול ביותר בקבוצה; משום ש–orderid הוא טור המפתח השני מייד לאחר orderdate.

תרשים 3–31: Ordered covering nonclustered index scan (תוכנית עבודה 2)



לפיכך, אין צורך שהתוכנית תמיין את הנתונים; אלא שהתוכנית רק אוספת אותם עם ordered scan מהאינדקס–המכסה, אשר כבר ממוין לפי orderdate ו–orderid. לאופרטור TOP יש משימה פשוטה שהיא רק לאסוף את השורה האחרונה (TOP 1 בסדר יורד), שהיא השורה בה מעוניינים בקבוצה. מספר השורות המדווחות על ידי האופרטור TOP הוא 1491, שהוא מספר הקבוצות הייחודיות (ערכי orderdate), שעבור כל אחת מהן האופרטור החזיר שורה בודדת. משום שהאינדקס–nonclustered שלנו מכסה את השאילתה על ידי כך שהוא כולל ברמת העלה שלו את כל הטורים המוזכרים בשאילתה (custid, empid), אין צורך לחפש את שורות הנתונים; השאילתה מקבלת תשובה על ידי נתוני האינדקס בלבד. להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 4720
- קריאות פיסיות: 3
- קריאות read-ahead: 4695
- זמן מעבד: 781 ms
- זמן שעון: 2128 ms
- עלות תת–עץ משוערת: 4.68047

מספר הקריאות הלוגיות שאתה רואה דומה למספר הדפים שמחזיקה רמת העלה של האינדקס.

Nonclustered Index Seek + Ordered Partial Scan + Lookups

שיטת הגישה nonclustered index seek + ordered partial scan + lookups משמשת לרוב לשאילתות תחומים–קטנים (כולל שאילתה נקודתית) המשתמשות באינדקס–nonclustered שאינו מכסה את השאילתה. כדי להדגים שיטת גישה זו, אשתמש בשאילתה הבאה:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid BETWEEN 101 AND 120;
```

אמנם יש לנו אינדקס הנקרא PK_Orders המוגדר על טור המפתח orderid, אך אינדקס זה אינו מכסה את השאילתה. אם השאילתה **סלקטיבית** מספיק, ה–optimizer ישתמש באינדקס. סלקטיביות מוגדרת כאחוז שמהווה מספר השורות המוחזרות על ידי השאילתה ממספר השורות הכולל בטבלה. המונח **סלקטיביות גבוהה** מתייחס לאחוז נמוך, בעוד המונח **סלקטיביות נמוכה** מתייחס לאחוז גבוה. שיטת הגישה שלנו מבצעת ראשית חיפוש בתוך האינדקס כדי למצוא את המפתח הראשון בתחום אותו מחפשים (orderid = 101). החלק השני של שיטת הגישה הוא ordered partial scan ברמת העלה מהמפתח הראשון בתחום ועד האחרון (orderid = 120). החלק השלישי והאחרון מערב lookups של שורת הנתונים המתאימה עבור כל מפתח. שים לב שהחלק השלישי אינו צריך לחכות שהחלק השני יסיים. לכל מפתח שנמצא בתחום, SQL Server יכול כבר להפעיל lookup. זכור ש–lookup ב–heap מתורגם לקריאת דף בודד, בעוד ש–lookup בטבלה–clustered מתורגם למספר קריאות כמספר הרמות באינדקס–clustered (3 במקרה שלנו).

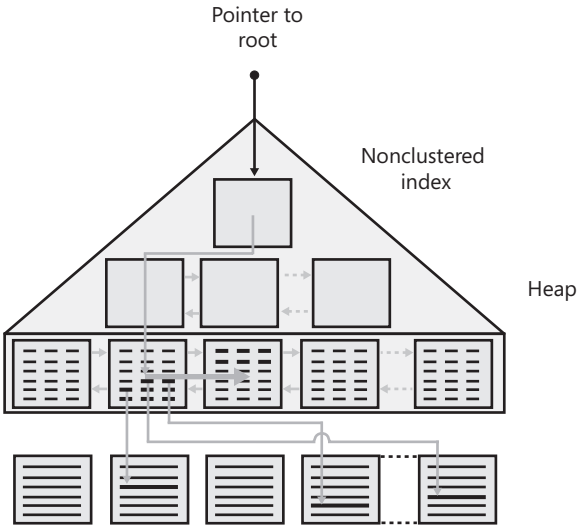
לצורך ביצוע הערכות ביצועים חיוני להבין שבשיטת גישה זו החלק האחרון המערב את ה–lookups לרוב מהווה את מרבית עלות השאילתה; זאת משום שהוא מערב את מרבית פעילות ה–I/O. זכור, ה–lookup מתורגם לקריאת דף מלאה או חיפוש אחד מלא בתוך האינדקס–clustered לכל שורה אותה מחפשים, וה–lookups הם תמיד I/O רנדומאלי (בניגוד לרציף).

לצורך הערכת עלות ה–I/O של שאילתה כזו, תוכל לרוב להתמקד בעלות של ה–lookups. אם ברצונך לבצע הערכות מדויקות יותר, ולקחת בחשבון גם את החיפוש בתוך האינדקס ואת ה– ordered partial scan, הרגש חופשי לעשות זאת; אך חלקים אלו יהיו זניחים ככל שהתחום גדל. עלות ה–I/O של פעולת חיפוש היא 3 קריאות במקרה שלנו, (מספר הרמות באינדקס). עלות ה–I/O של ה– ordered partial scan תלויה במספר השורות בתחום (20 במקרה שלנו). עבור השאילתה שלנו, אין למעשה שום קריאה נוספת מעורבת עבור הסריקה החלקית; זאת משום שכל המפתחות בתחום בו אנו מחפשים נמצאים בדף העלה אליו הגיע החיפוש, או שהם מתפרשים על דף נוסף אם המפתח הראשון מופיע קרוב לסוף הדף. עלות ה–I/O של פעולות ה–lookup יהיה מספר השורות בתחום (20 במקרה שלנו), מוכפל באחד אם הטבלה היא heap, או מוכפל במספר הרמות באינדקס–clustered

(3 במקרה שלנו) אם הטבלה היא clustered. כך שעליך לצפות לכ–23 קריאות לוגיות בסך הכל אם אתה מריץ את השאילתה מול heap וכ–63 קריאות לוגיות אם אתה מריץ אותה מול טבלה clustered. זכור שהרמות שאינן–עלה באינדקס–clustered נמצאות לרוב ב–cache בגלל כל פעולות ה–lookup שעוברות בו; כך שאל לך לדאוג יותר מדי כתוצאה ממה שנראה כעלות גבוהה יותר של שאילתה בתרחיש של טבלה–clustered.

תרשים 3–32 מציג שרטוט של שיטת הגישה על heap, ותרשים 3–33 מציג את תוכנית העבודה של השאילתה:

תרשים 3–32: Nonclustered index seek + ordered partial scan + lookups against a heap

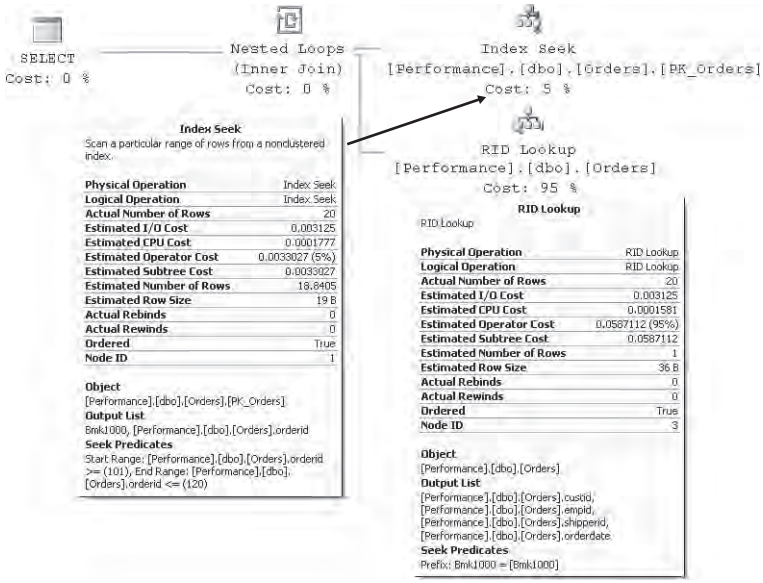


שים לב שבתוכנית העבודה לא תראה בצורה מפורשת את החלק partial scan של שיטת הגישה; אלא, הוא מתחבא באופרטור Index Scan. תוכל להסיק שהוא שם מהעובדה שתיבת המידע עבור האופרטור מראה True במדד Ordered.

להלן מדידות הביצועים שקיבלתי עבור השאילתה:

- קריאות לוגיות: 23
- קריאות פיסיות: 22
- זמן מעבד: 0 ms
- זמן שעון: 133 ms
- עלות תת–עץ משוערת: 0.0620926

תרשים 3–33: Nonclustered index seek + ordered partial scan + lookups against a heap (תוכנית עבודה)



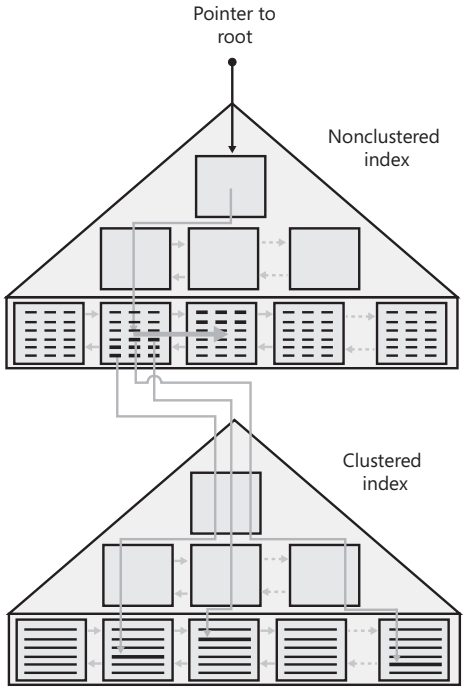
תרשים 3–34 מציג שרטוט של שיטת הגישה על טבלה–clustered, ותרשים 3–35 מציג את תוכנית העבודה של השאילתה.

להלן מדידות הביצועים שקיבלתי עבור השאילתה במקרה זה:

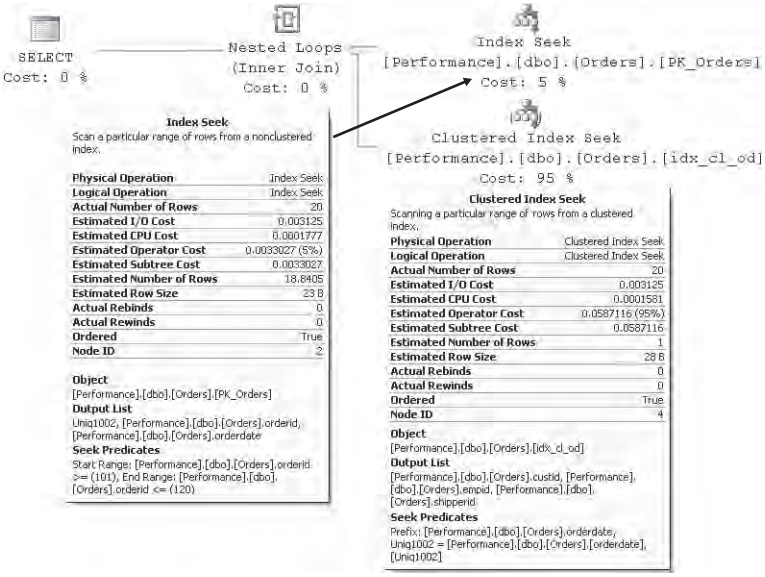
- קריאות לוגיות: 63
- קריאות פיסיות: 6
- זמן מעבד: 0 ms
- זמן שעון: 136 ms
- עלות תת–עץ משוערת: 0.0620931

באופן מעניין, תוכניות עבודה גרפיות ב– SQL Server 2000 לא ביצעו הבחנה ברורה בין RID lookup לבין clustering key lookup, כשהאחרון הוא למעשה חיפוש בתוך האינדקס–clustered. ב– SQL Server 2000, שניהם נקראו פשוט Bookmark Lookup. כפי שתוכל לראות בתוכניות המוצגות בתרשימים 3–33 ו–3–35, תוכניות גרפיות ב– SQL Server 2005 מראות את ההבדל בצורה מדויקת יותר.

תרשים 3–34: Nonclustered index seek + ordered partial scan + lookups against a clustered table



תרשים 3–35: Nonclustered index seek + ordered partial scan + lookups against a clustered table (תוכנית עבודה)



שיטת גישה זו יעילה רק כאשר השאילתה מאוד סלקטיבית (שאילתה נקודתית או תחום קטן). נסה לשחק עם התחום במסנן, הגדל אותו הדרגתית וראה כיצד העלות גדלה בצורה דרמטית ככל שהתחום גדל. דבר זה יתרחש עד לנקודה בה ה–optimizer יבין שיהיה זה פשוט יעיל יותר להפעיל table scan מאשר להשתמש באינדקס. אדגים תרגיל כזה בהמשך הפרק בסעיף "סרגל אופטימיזציה של אינדקס".

זכור שפעולות ממוינות, כמו החלק ordered partial scan של שיטת גישה זו, יכולות להתרחש הן קדימה והן אחורה. במקרה של השאילתה שלנו, היה זה קדימה, אך תוכל לחקור מקרים כמו מסנן על orderid <= 100, ולראות ordered backwards partial scan. אינדיקציה לשאלה האם הסריקה הייתה ממוינת קדימה או אחורה לא תופיע בתיבת המידע הצהובה של האופרטור; אלא היא תופיע במדד Scan Direction בתיבת הדו–שיח Properties של האופרטור.

Unordered Nonclustered Index Scan + Lookups

ה–optimizer לרוב משתמש בשיטת הגישה Unordered Nonclustered Index Scan + Lookups כאשר מתקיימים התנאים הבאים:

- השאילתה סלקטיבית מספיק.
- האינדקס האופטימלי לשאילתה לא מכסה אותה.
- האינדקס אינו מחזיק את המפתחות אחריהם מחפשים בצורה ממוינת.

למשל, כזה הוא המקרה כאשר אתה מסנן טור שאינו טור המפתח הראשון באינדקס. שיטת הגישה תערב סריקה מלאה לא ממוינת של רמת העלה של האינדקס, שלאחריה סדרת lookups. כפי שציינתי, על השאילתה להיות סלקטיבית מספיק כדי להצדיק שיטת גישה זו; אחרת, עם lookups רבים מדי יהיה זה יקר יותר מאשר פשוט לסרוק את הטבלה כולה. כדי לדעת מה מידת הסלקטיביות של השאילתה, SQL Server יזדקק לסטטיסטיקות על הטור המסונן (היסטוגרמה עם פיזור הערכים). אם סטטיסטיקות כאלו לא קיימות, SQL Server ייצור אותן.

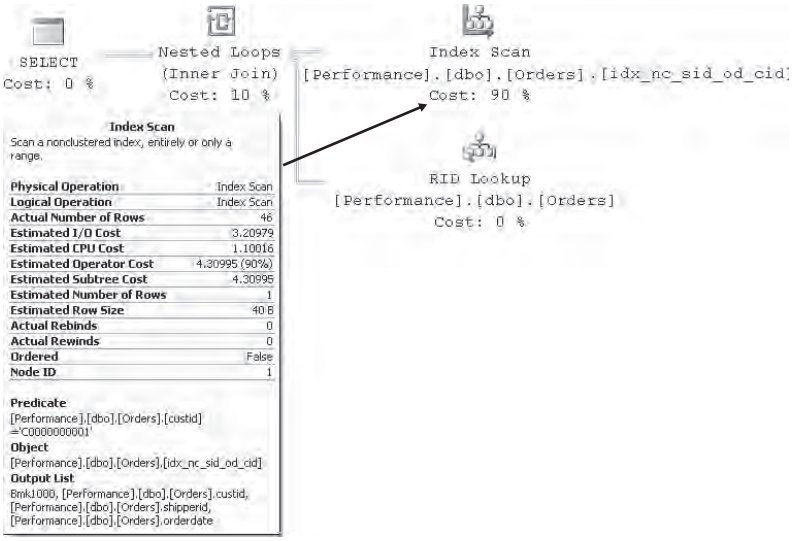
למשל, השאילתה הבאה תשתמש בשיטת גישה כזו מול האינדקס idx_nc_sid_od_cid, שנוצר על טור המפתח (shipperid, orderdate, custid), כאשר custid אינו טור המפתח הראשון ברשימה:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE custid = 'C0000000001';
```

תרשים 36–3 משרטט את שיטת הגישה על heap, ותרשים 37–3 מציג את תוכנית העבודה עבור השאילתה.

תרשים 3–36: Unordered nonclustered index scan + lookups against a heap

תרשים 3–37: Unordered nonclustered index scan + lookups against a heap
(תוכנית עבודה)



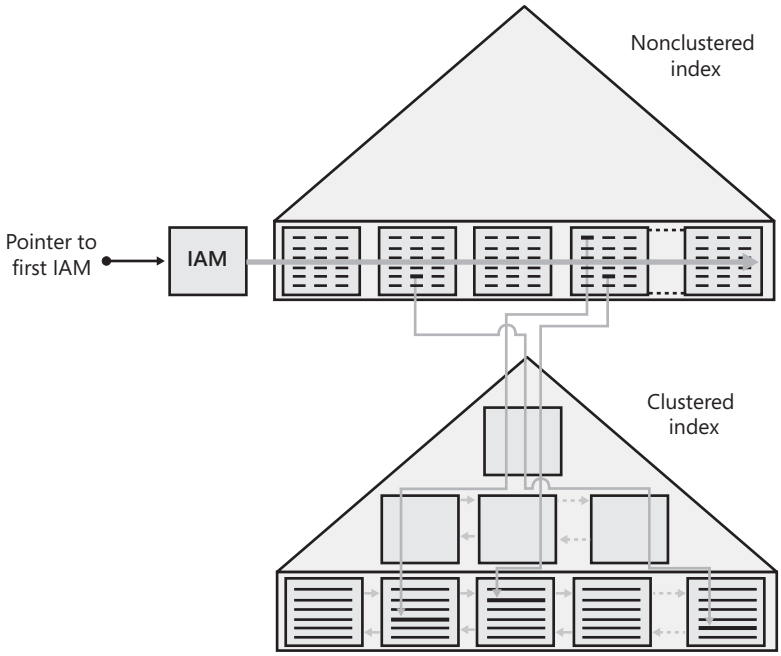
עלות ה–I/O של שאילתה זו כוללת את עלות ה– unordered scan של רמת העלה של האינדקס (I/O רציף תוך שימוש בדפי IAM) ועוד עלות ה–lookups (I/O רנדומאלי). הסריקה תעלה מספר קריאות דף כמספר הדפים ברמת העלה של האינדקס. כפי שתואר קודם, עלות ה–lookups היא מספר השורות העונות על השאילתה כפול 1 ב–heap, וכפול מספר הרמות באינדקס–clustered (3 במקרה שלנו) אם הטבלה היא clustered. להלן המדידות שקיבלתי עבור שאילתה זו על heap:

- קריאות לוגיות: 4400

- קריאות פיסיות: 47
- קריאות read-ahead: 4345
- זמן מעבד: 1281 ms
- זמן שעון: 2287 ms
- עלות תת-עץ משוערת: 4.479324

תרשים 3-38 משרטט את שיטת הגישה על טבלה-clustered, ותרשים 3-39 מציג את תוכנית העבודה של השאילתה.

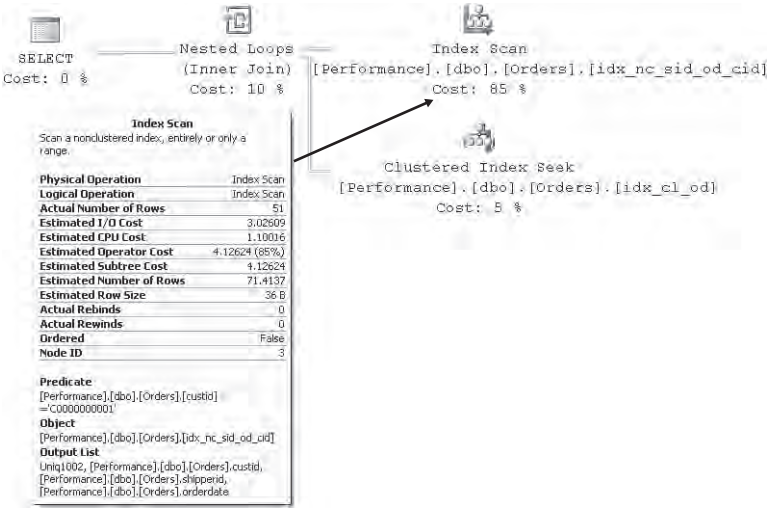
תרשים 3-38: Unordered nonclustered index scan + lookups against a clustered table



להלן המדידות שקיבלתי עבור שאילתה זו על טבלה-clustered:

- קריאות לוגיות: 4252
- קריאות פיסיות: 89
- קריאות read-ahead: 4090
- זמן מעבד: 1031 ms
- זמן שעון: 3148 ms
- עלות תת-עץ משוערת: 4.60953

תרשים 3–39: Unordered nonclustered index scan + lookups against a clustered table (תוכנית עבודה 1)



זכור ש– SQL Server יזדקק לסטטיסטיקות על הטור custid כדי לקבוע את מידת הסלקטיביות של השאילתה. השאילתה הבאה תאמר לך אילו סטטיסטיקות SQL Server ייצר אוטומטית על טבלת Orders:

```
SELECT name
FROM sys.stats
WHERE object_id = OBJECT_ID('dbo.Orders')
  AND auto_created = 1;
```

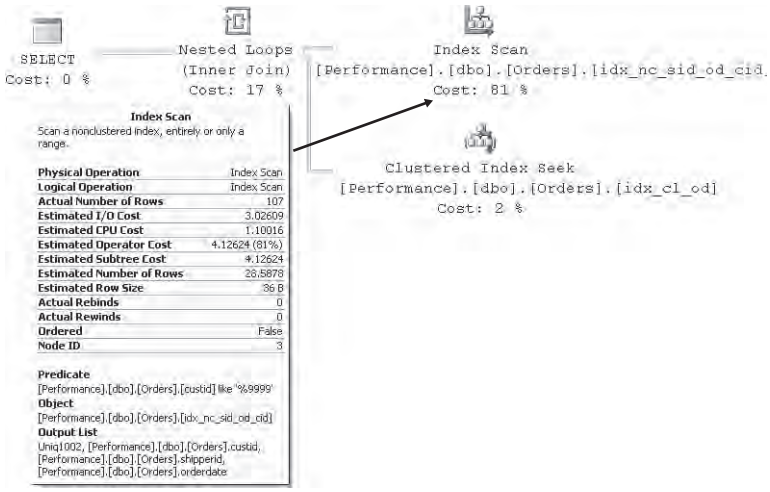
תקבל סטטיסטיקות עם שם דומה ל– _WA_Sys_00000002_31B762FC, ש– SQL Server ייצר אוטומטית למטרה זו.

SQL Server 2005 לראשונה מנהל מידע סטטיסטי על תת–מחרוזות בתוך טורים מטיפוס נתונים מחרוזתי. בהתבסס על מידע זה, יכול ה–optimizer לבצע הערכות סלקטיביות מדויקות יותר ולהפיק תוכניות עבודה יעילות יותר. כעת הוא יכול להעריך את הסלקטיביות של שאילתה כאשר אתה מפעיל מסננים מבוססי–תבנית עם הביטוי LIKE אפילו כאשר התבנית מתחילה בתו הכללה. יכולת זו לא הייתה קיימת בגרסאות מוקדמות יותר של SQL Server.

כדי להדגים יכולת זו, SQL Server יוכל להעריך את הסלקטיביות של השאילתה הבאה, המפיקה את התוכנית המוצגת בתרשים 3–40 על ידי שימוש בשיטת הגישה, שהיא מרכז הדיון בסעיף זה:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE custid LIKE '%9999';
```

תרשים 3-40: Unordered nonclustered index scan + lookups against a clustered table (תוכנית עבודה 2)



להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 4685
- קריאות פיסיות: 1
- קריאות read-ahead: 3727
- זמן מעבד: 3795 ms
- זמן שעון: 824 ms
- עלות תת-עץ משוערת: 5.09967

Clustered Index Seek + Ordered Partial Scan

ה-optimizer משתמש לרוב בשיטת הגישה clustered index seek + ordered partial scan לשאילתות תחום בהן אתה מסנן בהתבסס על טור המפתח הראשון של האינדקס-clustered. שיטת גישה זו מבצעת ראשית פעולת חיפוש למפתח הראשון בתחום; אז היא מפעילה ordered partial scan ברמת העלה מהמפתח הראשון בתחום ועד האחרון. היתרון העיקרי של שיטה זו הוא שלא מעורבים בה lookups. זכור ש-lookups יקרים מאוד בתחומים גדולים. יחס הביצועים בין שיטת גישה זו - שאינה כוללת lookups - וזו שמשתמשת באינדקס-nonclustered ו-lookups גדל יותר ויותר ככל שהתחום גדל.

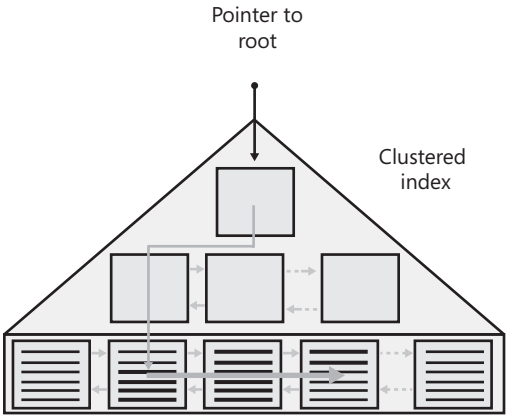
השאילתה הבאה, המחפשת אחר כל ההזמנות שבוצעו ב-orderdate מסוים, משתמשת בשיטת הגישה, שהיא מוקד דיון זה:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderdate = '20060212';
```

שים לב שלמרות שהמסנן משתמש באופרטור שוויון, זוהי במהותה שאילתת תחום משום שישנן מספר שורות שעונות על תנאי הסינון שלה. בכל מקרה, שאילתה נקודתית יכולה להיחשב כמקרה פרטי של שאילתת תחום. עלות ה–I/O של שיטת גישה זו תכלול את עלות פעולת החיפוש (3 קריאות רנדומאליות במקרה שלנו) ואת עלות ה– ordered partial scan בתוך העלה (במקרה שלנו, 18 קריאות דף). בסך הכל, אתה מקבל 21 קריאות לוגיות. שים לב שה– ordered scan לרוב אחראית לרוב עלות השאילתה משום שהיא מערבת את מרבית ה–I/O. זכור שעם ordered index scan, פרגמנטציה של אינדקסים ממלאת תפקיד משמעותי. כאשר הפרגמנטציה מינימלית (כמו במקרה שלנו), קריאות פיסיות יהיו כמעט רציפות. ככל שרמת הפרגמנטציה גדלה, זרוע הדיסק תהיה חייבת לזוז בצורה תזזיתית לפה ולשם, תוך שהיא מורידה את רמת הביצועים של הסריקה.

תרשים 3–41 משרטט את שיטת הגישה, ותרשים 3–42 מציג את תוכנית העבודה עבור השאילתה.

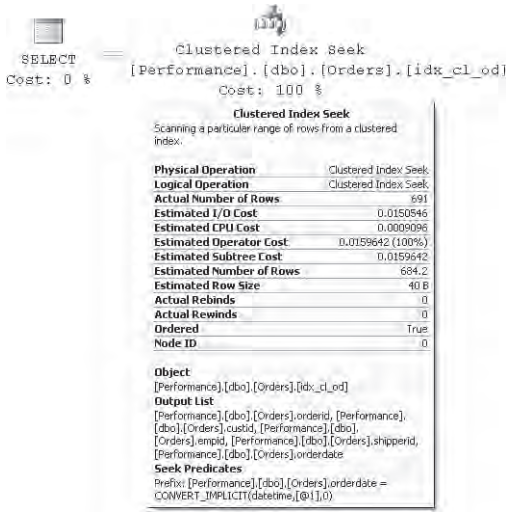
תרשים 3–41: Clustered index seek + ordered partial scan



להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 21
- קריאות פיסיות: 0
- קריאות read-ahead: 18
- זמן מעבד: 0 ms
- זמן שעון: 73 ms
- עלות תת–עץ משוערת: 0.0159642

תרשים 3-42: Clustered index seek + ordered partial scan (תוכנית עבודה)



שים לב שטריוויאלי ל-optimizer לייצר תוכנית זו. כלומר, התוכנית אינה תלויה בסלקטיביות של השאילתה. אלא, היא תמיד תהיה בשימוש בלי קשר לגודל התחום אחריו מחפשים. אלא אם כן, כמובן, יש לך מלכתחילה אינדקס אפילו טוב יותר לשאילתה.

Covering Nonclustered Index Seek + Ordered Partial Scan

שיטת הגישה covering nonclustered index seek + ordered partial scan זהה כמעט לחלוטין לשיטת הגישה שתוארה קודם, כאשר ההבדל היחיד הוא שזו משתמשת באינדקס-מכסה nonclustered- במקום באינדקס-clustered. כדי להשתמש בשיטה זו, כמובן שהטורים המסוננים חייבים להיות טורי המפתח הראשונים באינדקס. היתרון של שיטת גישה זו על פני הקודמת מונח בעובדה שדף עלה של אינדקס-nonclustered באופן טבעי יכול להכיל יותר שורות מאשר דף של אינדקס-clustered; לפיכך, מרבית עלות התוכנית, שהיא עלות הסריקה החלקית של העלה, נמוכה יותר. העלות נמוכה יותר משום שיש צורך לסרוק פחות דפים עבור אותו גודל תחום. כמובן, גם כאן, פרגמנטציה של אינדקסים ממלאת תפקיד חשוב בהקשר של ביצועים משום שהסריקה החלקית ממוינת.

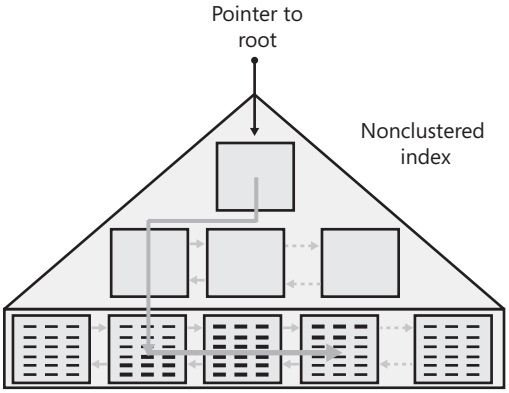
כדוגמה, השאילתה הבאה המחפשת תחום ערכי orderdate ל-shipperid נתון, משתמשת בשיטת גישה זו מול האינדקס המכסה idx_nc_sid_od_cid, שנוצר על (shipperid, orderdate, custid):

```
SELECT shipperid, orderdate, custid
FROM dbo.Orders
WHERE shipperid = 'C'
  AND orderdate >= '20060101'
  AND orderdate < '20070101';
```

שים לב: כדי לגרום לסריקה החלקית לקרוא את הכמות המינימלית הנדרשת של דפים, טורי המפתח הראשונים של האינדקס חייבים להיות shipperid, orderdate, בסדר זה (משמאל לימין). אם תחליף את הסדר, הסריקה החלקית תסרוק בסופו של דבר גם שורות הנכללות בתחום התאריכים גם למובילים אחרים, מה שידרוש יותר I/O.

תרשים 3–43 משרטט את שיטת הגישה, ותרשים 3–44 מציג את תוכנית העבודה של השאילתה.

תרשים 3–43: Covering nonclustered index scan + ordered partial scan

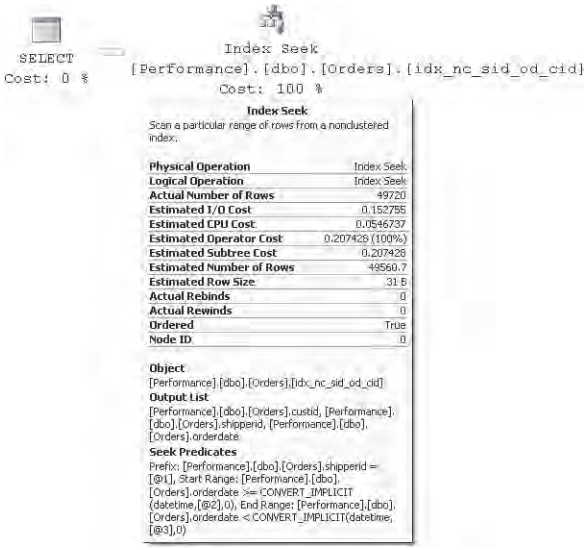


להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 208
- זמן מעבד: 30 ms
- זמן שעון: 954 ms
- עלות תת–עץ משוערת: 0.207428

שים לב שתוכנית זו גם היא תוכנית טריוויאלית שאינה מבוססת על סלקטיביות השאילתה.

תרשים 3-44: Covering nonclustered index scan + ordered partial scan (תוכנית עבודה)



זכור, היתרון העיקרי של שיטת גישה זו היא שכלל לא מעורבים lookups משום שהאינדקס מכסה את השאילתה. כמו כן, אתה קורא פחות דפים מאשר בשיטת גישה דומה מול אינדקס-clustered.

כמו כן שים לב שכאשר אתה יוצר אינדקסים מכסים, טורי האינדקס ממלאים שני תפקידים שונים. טורים שלפיהם אתה מסנן או ממיין נדרשים כטורי מפתח שיוחזקו בכל רמות העץ המאוזן, והם גם יקבעו את סדר המיון ברמת העלה. טורי אינדקס אחרים עשויים להידרש רק למטרות כיסוי. אם אתה כולל את כל טורי האינדקס ברשימת טורי המפתח של האינדקס, עליך לזכור שיש לכך מחיר. SQL Server צריך לשמור את העץ מאוזן, ויהיה עליו לגרום לתנועה פיסית של הנתונים ולשינויים בעץ כאשר אתה משנה ערכי טור מפתח בטבלה. זהו בזבוז עם טורים הנדרשים רק למטרות כיסוי ולא לסינון או למיון.

כדי למלא צורך זה, SQL Server 2005 מציג את המושג included non-key columns באינדקס. כאשר אתה יוצר אינדקס, אתה מגדיר בנפרד אילו טורים יהוו את רשימת המפתחות ואילו יכללו רק למטרות כיסוי-רק ברמת העלה של האינדקס.

כדוגמה, השאילתה האחרונה שלנו נשענה רק על shipperid ו-orderdate למטרות סינון ומיון, בעוד שהיא נשענה על custid רק למטרות כיסוי. כדי ליהנות מהמאפיין החדש ב- SQL server 2005, הסר את האינדקס וצור אחד חדש, כשאתה מגדיר custid בפסוקית INCLUDE כלהלן:

```
DROP INDEX dbo.Orders.idx_nc_sid_od_cid;

CREATE NONCLUSTERED INDEX idx_nc_sid_od_i_cid
  ON dbo.Orders(shipperid, orderdate)
  INCLUDE(custid);
```

שים לב שרשימת המפתחות מוגבלת ל–16 טורים ו–900 בתים. דבר זה נכון הן ב– SQL server 2000 והן ב– SQL Server 2005. בונוס נוסף של טורי included non-key הוא שהם אינם כפופים לאותן מגבלות כמו טורי המפתח. למעשה, הם יכולים אפילו לכלול אובייקטים גדולים כמו טורים בעלי אורך משתנה המוגדרים עם האפשרות MAX וטורי XML.

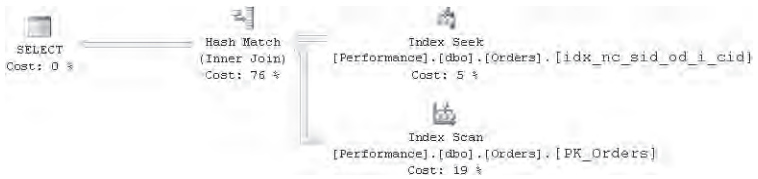
הצלבת אינדקסים (Index Intersection)

עד כה, התמקדתי בעיקר ביתרון הביצועים שאתה מקבל מאינדקסים בעת קריאת נתונים. עם זאת זכור, שלאינדקסים יש עלות כאשר אתה מעדכן נתונים. כל שינוי של נתונים (מחיקות, הוספות ועדכונים) חייב להשתקף באינדקסים המחזיקים עותק של נתונים אלה. הדבר עלול לגרום לפיצולי דפים ולשינויים בעצים המאוזנים, שעשויים להיות יקרים מאוד. לפיכך, אינך יכול ליצור כמה אינדקסים שאתה רוצה בצורה חופשית, במיוחד במערכות אשר יש בהן עדכונים אינטנסיביים כמו מערכות OLTP. תרצה להכין סדר עדיפויות ולבחור באינדקסים החשובים יותר. זוהי בעיה במיוחד עם אינדקסים מכסים משום ששאילתות שונות יכולות להפיק תועלת מאינדקסים מכסים שונים לחלוטין, ואתה עלול בסופו של דבר לקבל מספר גדול מאוד של אינדקסים מהן השאילתות שלך יכולות להפיק תועלת.

למזלנו, הבעיה מעט פחות גדולה משום שה–optimizer תומך בטכניקה הנקראת הצלבת אינדקסים, בה הוא מצליב נתונים שהתקבלו משני אינדקסים ואם נדרש, מצליב את התוצאה עם נתונים שהתקבלו באינדקס נוסף, וכך הלאה. כדוגמה, ה–optimizer ישתמש בהצלבת אינדקסים עבור השאילתה הבאה, ויפיק את התוכנית המוצגת בתרשים 45–3:

```
SELECT orderid, custid
FROM dbo.Orders
WHERE shipperid = 'A';
```

תרשים 45–3: תוכנית עבודה עם הצלבת אינדקסים



בנושא אופרטורי join ארחיב את הדיבור בפרק 5. האינדקס האופטימלי כאן יהיה כזה בו shipperid מוגדר כטור המפתח, ו–orderid ו–custid מוגדרים כטורים מוכללים שאינם מפתח, אך לא קיים אינדקס כזה על הטבלה. מה שקיים הוא האינדקס idx_nc_sid_od_i_cid המגדיר shipperid כטור המפתח וכולל גם את הטור custid, והאינדקס PK_Orders המכיל את הטור orderid. ה–optimizer השתמש בשיטת הגישה nonclustered index seek + ordered partial scan כדי להשיג את הנתונים הרלוונטיים מ–idx_nc_sid_od_i_cid, ובשיטת הגישה unordered nonclustered index scan כדי להשיג את הנתונים הרלוונטיים מ–PK_Orders. הוא אז מצליב את שני הסטים בהתבסס על ערכי מאתרי השורה; באופן טבעי, ערכי מאתרי השורה המצביעים לאותן שורות ימצאו התאמה. ניתן לחשוב על הצלבת אינדקסים כ–join פנימי בהתבסס על התאמה בערכי מאתרי שורה.

להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- מספר סריקות: 2
- קריאות לוגיות: 3671
- קריאות פיסיות: 33
- קריאות read-ahead: 2347
- זמן מעבד: 1161 ms
- זמן שעון: 5202 ms
- עלות תת–עץ משוערת: 16.7449

Indexed Views

סעיף זה מתאר ומדגים בקצרה את המושג indexed views לצורך שלמות הדיון. לא אקיים כאן דיון מעמיק בנושא. פרטים נוספים אספק בספר **תכנות T-SQL** ואכוון אותך למקורות נוספים בסוף הפרק.

מאז SQL Server 2000, באפשרותך ליצור אינדקסים על views – לא רק על טבלאות. View הוא אובייקט וירטואלי, ושאילתה מולו בסופו של דבר מתבצעת על הטבלאות שמאחוריו. עם זאת, כאשר אתה יוצר אינדקס–clustered על view אתה מממש את כל תכני ה–view בתוך האינדקס–clustered על הדיסק. לאחר שאתה יוצר את האינדקס–clustered, באפשרותך ליצור גם מספר אינדקסים–noncluetred על ה–view. הנתונים באינדקסים על ה–view יישמרו מסונכרנים עם השינויים בטבלאות שמאחורי ה–view כמו בכל אינדקס אחר.

יתרונם של indexed views הוא בעיקר בכך שהם מפחיתים עלויות I/O ועיבוד יקר של נתונים. עלויות כאלו בולטות בעיקר בשאילתות צבירה הסורקות נפחים גדולים של נתונים ומפיקות סטי תוצאה קטנים, ובשאילתות join יקרות.

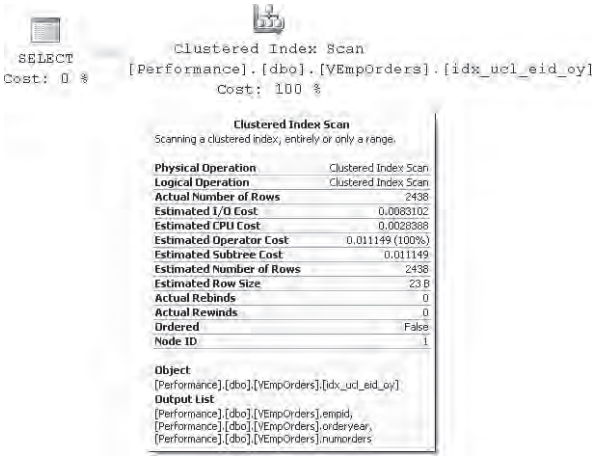
כדוגמה, הקוד הבא יוצר indexed views המעוצב לכוונון שאילתות צבירה המקבצות הזמנות לפי empid ו–YEAR(orderdate), ומחזירות את מספר ההזמנות בכל קבוצה:

```
IF OBJECT_ID('dbo.VEmpOrders') IS NOT NULL
  DROP VIEW dbo.VEmpOrders;
GO
CREATE VIEW dbo.VEmpOrders
  WITH SCHEMABINDING
AS

SELECT empid, YEAR(orderdate) AS orderyear, COUNT_BIG(*) AS numorders
FROM dbo.Orders
GROUP BY empid, YEAR(orderdate);
GO

CREATE UNIQUE CLUSTERED INDEX idx_ucl_eid_oy
  ON dbo.VEmpOrders(empid, orderyear);
```

בצע שאילתה על ה–view, ותקבל את תוכנית העבודה המוצגת בתרשים 3–46, המראה שהאינדקס–clustered על ה–view נסרק:

```
SELECT empid, orderyear, numorders
FROM dbo.VEmpOrders;
```

תרשים 3–46: תוכנית עבודה לשאילתה על indexed view



ה–view מכיל מספר קטן מאוד של שורות (בסביבות מספר אלפים) בהשוואה למספר השורות בטבלה (מיליון). רמת העלה של האינדקס מכילה רק כ–10 דפים. לפיכך, עלות ה–I/O של התוכנית תהיה בערך 10 קריאות דף.

להלן מדידות הביצועים שקיבלתי עבור השאילתה:

- קריאות לוגיות: 10
- זמן מעבד: 0 ms
- זמן שעון: 78 ms
- עלות תת–עץ משוערת: 0.011149

באופן מעניין, אם אתה עובד בגרסת Enterprise (או Developer) של SQL Server, ה–optimizer ישקול שימוש באינדקסים על view אפילו כאשר מבצעים שאילתות ישירות על הטבלאות שמאחוריו. למשל, השאילתה הבאה מייצרת תוכנית דומה לזו שמוצגת בתרשים 3–46, עם אותה עלות שאילתה:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT_BIG(*) AS numorders
FROM dbo.Orders
GROUP BY empid, YEAR(orderdate);
```

אם אינך עובד בגרסת Enterprise, עליך לבצע את השאילתה ישירות על ה–view, וכן להגדיר שאינך רוצה שה–optimizer ירחיב את אפשרויות האופטימיזציה שלו מעבר לטווח ה–view. אתה עושה זאת על ידי הגדרת ה– NOEXPAND hint: FROM <view_name> WITH (NOEXPAND).

SQL Server 2005 מרחיב את הנסיבות בהן יש שימוש ב– indexed views, ותומך בתת–אינטרוולים, ביטויים מקבילים לוגית ועוד. כפי שציינתי, אכוון אותך למקורות המתארים את אלו בפירוט.

סרגל לאופטימיזציה של אינדקסים

היזכר בדיון הקודם על מתודולוגית כוונון. כאשר אתה מבצע כוונון לאינדקסים, אתה עושה זאת ביחס לתבניות השאילתה המהוות את העלות המצטברת הגבוהה ביותר במערכת. עבור תבנית שאילתה נתונה, באפשרותך לבנות סרגל אופטימיזציה של אינדקסים שיסייע לך לבצע את הבחירות הנכונות. אציג תהליך זה דרך דוגמה. כדי לעקוב אחר ההדגמות, בטרם אתה ממשיך, הסר את ה–view שייצרנו קודם ואת כל האינדקסים על טבלת Orders, למעט האינדקס–clustered. לחלופין, תוכל להריץ מחדש את הקוד בקטע–קוד 3–1, לאחר שינוי או הסרה של כל משפטי היצירה של אינדקסים ושל מפתח ראשי על Orders, תוך שמירה רק על האינדקס–clustered.

בדוגמה שלנו, נניח שעליך לכוונן את תבנית השאילתה הבאה:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= value;
```

זכור שהיעילות של כמה שיטות גישה תלויה בסלקטיביות של השאילתה, בעוד שהיעילות של אחרות אינה תלויה. עבור שיטות גישה התלויות בסלקטיביות, הנח שתבנית השאילתה היא לרוב סלקטיבית למדי (בסביבות 0.1 אחוז סלקטיביות, או בסביבות 1000 שורות שעונות על השאילתה). השתמש בשאילתה הבאה בתהליך הכוונון שלך כאשר אתה מכוון לסלקטיביות כזו:

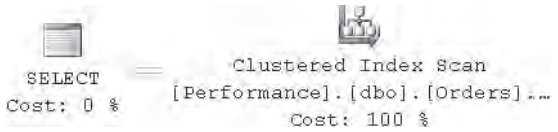
```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= 999001;
```

אתקדם בסרגל האופטימיזציה של אינדקסים מהתרחיש הגרוע ביותר לתרחיש הטוב ביותר, תוך שימוש בשאילתה זו כמקור התייחסות, אך אתאר גם מה יקרה כאשר סלקטיביות השאילתה תשתנה.

Table Scan (Unordered Clustered Index Scan)

התרחיש הגרוע ביותר עבור תבנית השאילתה שלנו עם סלקטיביות גבוהה למדי הוא כאשר אין לך שום אינדקס טוב. תקבל את תוכנית השאילתה המוצגת בתרשים 3–47, תוך שימוש ב– table scan (unordered clustered index scan).

תרשים 3–47: תוכנית עבודה עם table scan (unordered clustered index scan)



על אף שאתה מעוניין במספר נמוך יחסית של שורות (1000 במקרה שלנו), נסרקת כל הטבלה. אלו מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 25080
- זמן מעבד: 1472 ms
- זמן שעון: 16399 ms
- עלות תת–עץ משוערת: 19.6211

תוכנית זו היא טריוויאלית ואינה תלויה בסלקטיביות – כלומר, תקבל את אותה תוכנית ללא קשר לסלקטיביות השאילתה.

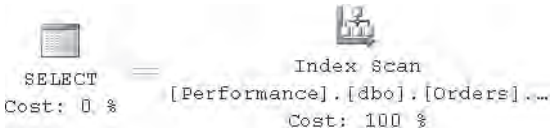
Unordered Covering Nonclustered Index Scan

הצעד הבא בסרגל האופטימיזציה יהיה ליצור אינדקס–מכסה–nonclustered כאשר הטור המסונן (orderid) אינו טור האינדקס הראשון:

```
CREATE NONCLUSTERED INDEX idx_nc_od_i_oid_cid_eid_sid
  ON dbo.Orders(orderdate)
  INCLUDE(orderid, custid, empid, shipperid);
```

אינדקס זה יניב שיטת גישה המשתמשת ב– unordered scan מלאה של רמת עלה האינדקס כפי שמוצג בתרשים 48–3:

תרשים 48–3: תוכנית עבודה עם unordered covering nonclustered index scan



גודל השורה באינדקס–מכסה זה היא בערך חמישית מגודל שורת נתונים מלאה, ודבר זה ישתקף בעלות השאילתה ובזמן הריצה שלה. להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- **קריאות לוגיות:** 5095
- **זמן מעבד:** 170 ms
- **זמן שעון:** 1128 ms
- **עלות תת–עץ משוערת:** 4.86328

כמו בתוכנית הקודמת, גם תוכנית זו היא טריוויאלית ואינה תלויה בסלקטיביות.

שים לב: זמן הריצה שתקבל עבור השאילתות שלך ישתנה בהתאם לכמות הנתונים הנמצאת ב–cache. אם ברצונך לערוך השוואות ביצועים אמינות במונחים של זמן ריצה, ודא שסביבת ה–cache בשני המקרים משקפת את מה שיהיה לך בסביבה התפעולית שלך. כלומר, אם אתה צופה שמרבית הדפים יהיו ב–cache בסביבה התפעולית שלך (cache חם), הרץ כל שאילתה פעמיים, ומדוד את זמן הריצה של ההרצה השנייה. אם אתה צופה שמרבית הדפים לא יהיו ב–cache (cache קר), נקה את ה–cache לפני שאתה מריץ כל שאילתה.

בטרם אתה ממשיך, הסר את האינדקס שזה עתה יצרת:

```
DROP INDEX dbo.Orders.idx_nc_od_i_oid_cid_eid_sid;
```

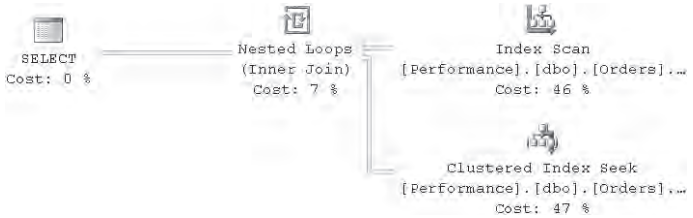
Unordered Nonclustered Index Scan + lookups

הצעד הבא בסרגל האופטימיזציה של אינדקסים שלנו הוא ליצור אינדקס–nonclustered קטן יותר שלא יכסה את השאילתה ואשר מכיל את הטור המסונן (orderid), אך לא כטור המפתח הראשון:

```
CREATE NONCLUSTERED INDEX idx_nc_od_i_oid
  ON dbo.Orders(orderdate)
  INCLUDE(orderid);
```

תקבל unordered nonclustered index scan + lookups כפי שנראה בתרשים 3–49.

תרשים 3–49: תוכנית עבודה עם unordered nonclustered index scan + lookups



שים לב שיעילות שאילתה זו בהשוואה לקודמת תהיה תלויה בסלקטיביות השאילתה. ככל שסלקטיביות השאילתה קטנה (אחוז השורות המוחזרות גדל), כך הופכת עלות ה–lookups כאן למשמעותית יותר. במקרה שלנו, השאילתה סלקטיבית למדי, כך שתוכנית זו יעילה יותר מאשר השתיים הקודמות; עם זאת, עם סלקטיביות נמוכה, תוכנית זו תהיה יעילה פחות מאשר השתיים הקודמות.

להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 5923
- זמן מעבד: 100 ms
- זמן שעון: 379 ms
- עלות תת–עץ משוערת: 7.02136

שים לב שלמרות שנראה שמספר הקריאות הלוגיות ועלות השאילתה גבוהים יותר מאשר בתוכנית הקודמת, זמני הריצה נמוכים יותר. זכור שפעולות ה–lookup כאן מבצעות פעולות חיפוש באינדקס–clustered, וסביר להניח שהרמות שאינן–עלה של האינדקס–clustered נמצאות ב–cache.

בטרם תמשיך, הסר את האינדקס החדש:

```
DROP INDEX dbo.Orders.idx_nc_od_i_oid;
```

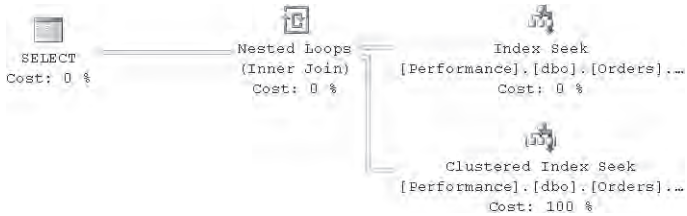
Nonclustered Index Seek + Ordered Partial Scan + Lookups

תוכל לקבל את הצעד הבא של אופטימיזציה בסרגל על ידי יצירת אינדקס–nonclustered שאינו מכסה על orderid:

```
CREATE UNIQUE NONCLUSTERED INDEX idx_unc_oid
  ON dbo.Orders(orderid);
```

האינדקס יניב nonclustered index seek + ordered partial scan + lookups כפי שמוצג בתרשים 50–3.

תרשים 50–3: תוכנית עבודה עם nonclustered index seek + ordered partial scan + lookups



במקום לבצע את ה– index scan המלאה כפי שעשתה התוכנית הקודמת, תוכנית זו מבצעת חיפוש למפתח הראשון בתחום המבוקש, ולאחריו ordered partial scan רק של התחום הרלוונטי. עדיין, אתה מקבל אותה כמות lookups כמקודם, שבמקרה שלנו מהווה חלק גדול מעלות השאילתה. ככל שהתחום גדל, התרומה של ה–lookups לעלות השאילתה הופכת להיות משמעותית יותר, והעלויות של שתי תוכניות אלה תהינה דומות יותר ויותר.

להלן מדידות הביצועים לשאילתה זו:

- קריאות לוגיות: 3077
- זמן מעבד: 0 ms
- זמן שעון: 53 ms
- עלות תת–עץ משוערת: 3.22852

קביעת נקודת הסלקטיביות

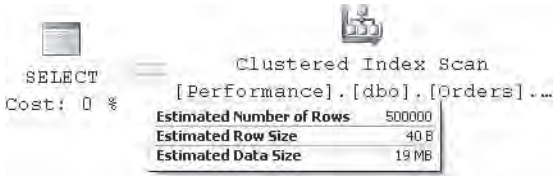
תרשו לי לסטות מעט כדי להרחיב את הדיבור על נושא בו התחלתי לדון קודם–תוכניות התלויות בסלקטיביות השאילתה. יעילות התוכנית האחרונה

תלויה בסלקטיביות משום שאתה מקבל lookup אחד מלא לכל שורה אותה מחפשים. בנקודת סלקטיביות מסוימת, ה–optimizer יבין ש– table scan יעילה יותר מאשר שימוש בתוכנית זו. ייתכן שזה יפתיע אותך, אך נקודת הסלקטיביות היא אחוז קטן למדי. גם אם אין שמץ של מושג כיצד לחשב נקודה זו, תוכל לתרגל גישת ניסוי וטעייה, כאשר אתה מפעיל אלגוריתם בינארי, ומזיז את נקודת הסלקטיביות שמאלה וימינה בהתבסס על התוכנית שאתה מקבל. זכור שסלקטיביות גבוהה משמעותה אחוז נמוך של שורות, כך שתזוזה שמאלה מנקודת הסלקטיביות (הקטנת האחוז) משמעותה תהיה קבלת סלקטיביות גבוהה יותר. תוכל לקרוא לשאילתת תחום, כאשר אתה מתחיל עם 50 אחוז סלקטיביות על ידי קריאה לשאילתה הבאה:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= 500001;
```

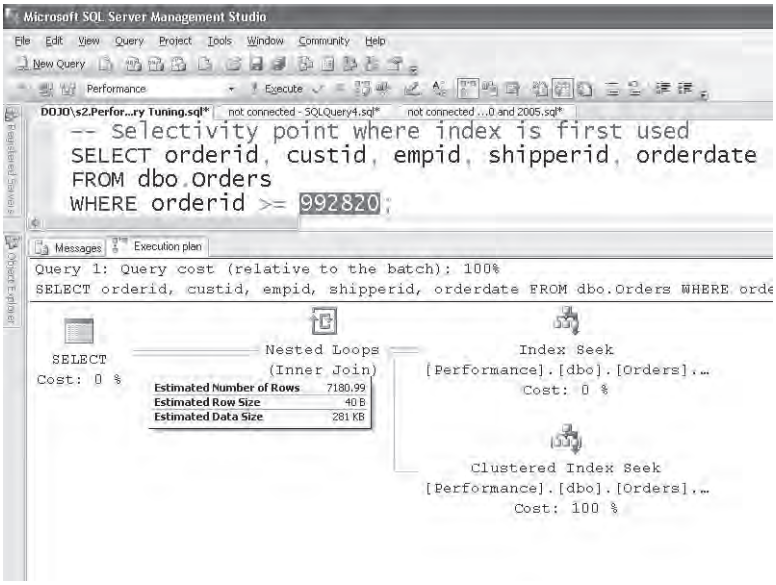
בחן את תוכנית העבודה המשוערת (אין צורך בתוכנית בפועל כאן), וקבע האם להמשיך בצעד הבא לשמאל או לימין נקודה זו, בהתבסס על מה שקיבלת – table scan (clustered index scan) או index seek. עם מפתח החציון, תקבל את תוכנית העבודה המוצגת בתרשים 3–51, המראה table scan:

תרשים 3–51: תוכנית משוערת המציגה table scan



דבר זה אומר לך ש–50 אחוז אינו מספיק סלקטיבי בשביל להצדיק שימוש באינדקס–nonclustered. אם כך אתה זז לימין, לנקודת האמצע בין 50 אחוז ו–100 אחוז. אם תמשיך לפי לוגיקה זו, תקבל לבסוף את המפתחות הבאים: 75001, 875001, 937501, 968751, 984376, 992189, 996095. המפתח האחרון יניב תוכנית בה יש שימוש באינדקס–nonclustered. כעת אתה זז לשמאל, לנקודה בין המפתחות 992189 ו–996095, שהיא 994142. תמצא שהאינדקס–nonclustered עדיין נמצא בשימוש, כך שאתה ממשיך לזוז שמאלה, עד לנקודה בין המפתחות 992189 ו–994142. אתה ממשיך בתהליך זה, כשאתה זז שמאלה או ימינה בהתאם לממצאיך, עד שאתה מגיע לנקודת הסלקטיביות הראשונה בה יש שימוש באינדקס–nonclustered. תמצא שנקודה זו היא המפתח 992820, המייצר את תוכנית העבודה המוצגת בתרשים 3–52:

תרשים 52–3: תוכנית משוערת המראה שהאינדקס בשימוש



כעת באפשרותך לחשב את הסלקטיביות, שהיא מספר השורות העונות על השאילתה (7181) חלקי מספר השורות בטבלה (1,000,000), כלומר 0.7181 אחוז.

במקרה של תבנית השאילתה שלנו, עם סלקטיביות זו או גבוהה יותר (אחוז נמוך יותר), ה–optimizer ישתמש באינדקס–nonclustered, בעוד שעם סלקטיביות נמוכה יותר, הוא יעדיף table scan. כפי שניתן לראות, במקרה של תבנית השאילתה שלנו, נקודת הסלקטיביות אפילו נמוכה מאחוז אחד. ייתכן שישנם מקצועני מסדי נתונים שיופתעו מכך שהמספר כל כך קטן, אך אם אתה מבצע הערכות ביצועים כמו אלו שביצענו קודם, תמצא שהוא סביר. אל תשכח שקריאות דפים אינו הגורם היחיד אותו עליך לקחת בחשבון. עליך לקחת בחשבון גם את דפוס הגישה (רנדומאלי/רציף) וגם גורמים אחרים. זכור ש–I/O רנדומאלי יקר הרבה יותר מאשר I/O רציף. Lookups משתמשים ב–I/O רנדומאלי, בעוד ש– table scan משתמשת ב–I/O רציף.

בטרם תמשיך, הסר את האינדקס בו השתמשנו בשלב הקודם:

```
DROP INDEX dbo.Orders.idx_unc_oid;
```

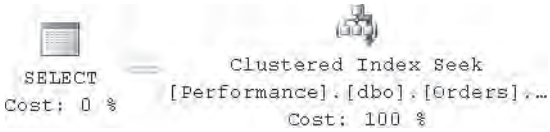
Clustered Index Seek + Ordered Partial Scan

תוכל לקבל את רמת האופטימיזציה הבאה על ידי יצירת אינדקס–clustered על הטור orderid. משום שיש כבר אינדקס–clustered על טבלת Orders, ראשית הסר אותו ואז צור את האינדקס הרצוי:

```
DROP INDEX dbo.Orders.idx_cl_od;
CREATE UNIQUE CLUSTERED INDEX idx_cl_oid ON dbo.Orders(orderid);
```

תקבל את התוכנית הטריוויאלית המשתמשת בחיפוש אחר המפתח הראשון המתאים למסנן, ולאחריו ordered partial scan של התחום המבוקש, כפי שמוצג בתרשים 53–3:

תרשים 53–3: תוכנית עבודה עם clustered index seek + ordered partial scan



היתרון העיקרי של תוכנית זו היא שאין בה lookups כלל. ככל שסלקטיביות השאילתה קטנה, תוכנית זו הופכת יותר ויותר יעילה בהשוואה לתוכנית המשתמשת ב–lookups. עלות ה–I/O המעורבת בתוכנית זו היא העלות של החיפוש (3 במקרה שלנו), ועוד מספר הדפים המחזיקים את שורות הנתונים בתחום המסונן (26 במקרה שלנו). העלות העיקרית של תוכנית כזו היא לרוב, עלות ה– ordered partial scan, אלא אם כן התחום הוא ממש קטנטן (למשל, שאילתה נקודתית). זכור שהביצועים של ordered index scan יהיו, במידה רבה, תלויים ברמת הפרגמנטציה של האינדקס. להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 29
- זמן מעבד: 0 ms
- זמן שעון: 87 ms
- עלות תת–עץ משוערת: 0.022160

לפני שתמשיך לצעד הבא, שחזר את האינדקס–clustered המקורי:

```
DROP INDEX dbo.Orders.idx_cl_oid;
CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);
```

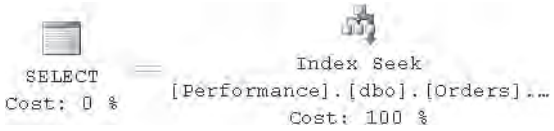
Covering Nonclustered Index Seek + Ordered Partial Scan

הרמה האופטימלית בסרגל שלנו היא nonclustered covering index המוגדר עם הטור orderid כמפתח וכל יתר הטורים כטורים מוכללים שאינם מפתח:

```
CREATE UNIQUE NONCLUSTERED INDEX idx_unc_oid_i_od_cid_eid_sid
  ON dbo.Orders(orderid)
  INCLUDE(orderdate, custid, empid, shipperid);
```

הלוגיקה של התוכנית דומה לזו הקודמת, אלא שכאן ה– ordered partial scan קוראת פחות דפים. זאת, כמובן, משום שיותר שורות נכנסות בדף עלה של אינדקס זה מאשר שורות נתונים בדף של אינדקס–clustered. אתה מקבל את התוכנית המוצגת בתרשים 54–3.

תרשים 54–3: תוכנית עבודה עם covering nonclustered index seek + ordered partial scan



ולהלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 9
- זמן מעבד: 0 ms
- זמן שעון: 47 ms
- עלות תת–עץ משוערת: 0.008086

שוב, זוהי תוכנית טריוויאלית. וגם כאן, הביצועים של ה– ordered partial scan ישתנו בהתאם לרמת הפרגמנטציה של האינדקס. כפי שתוכל לראות, עלות השאילתה ירדה מ–19.621100 ברמה הנמוכה ביותר בסרגל ל–0.008086, וזמן השעון ירד מלמעלה מ–16 שניות ל–47 אלפיות שנייה. כזו ירידה בזמן ריצה נפוצה כאשר מכוונים אינדקסים בסביבה בעלת עיצוב אינדקסים דל.

כאשר תסיים, הסר את האינדקס האחרון שיצרת:

```
DROP INDEX dbo.Orders.idx_unc_oid_i_od_cid_eid_sid;
```

סיכום וניתוח סרגל אופטימיזציה של אינדקסים

זכור שיעילות מספר תוכניות בסרגל האופטימיזציה של אינדקסים שלנו התבססה על סלקטיביות השאילתה. אם הסלקטיביות של שאילתה אותה אתה מכוונן משתנה בצורה משמעותית בין הרצות של השאילתה, ודא שבתהליך הכוונון שלך אתה לוקח זאת בחשבון. למשל, תוכל להכין טבלאות וגרפים עם מדידות הביצועים מול הסלקטיביות ולנתח נתונים אלו לפני שאתה מבצע בחירות בנוגע לעיצוב האינדקסים שלך. למשל, טבלה 16–3 מציגה סיכום של קריאות לוגיות מול סלקטיביות עבור תבנית השאילתה לדוגמה בה דנו. זאת עבור כל אחת משיטות הגישה בסרגל האופטימיזציה.

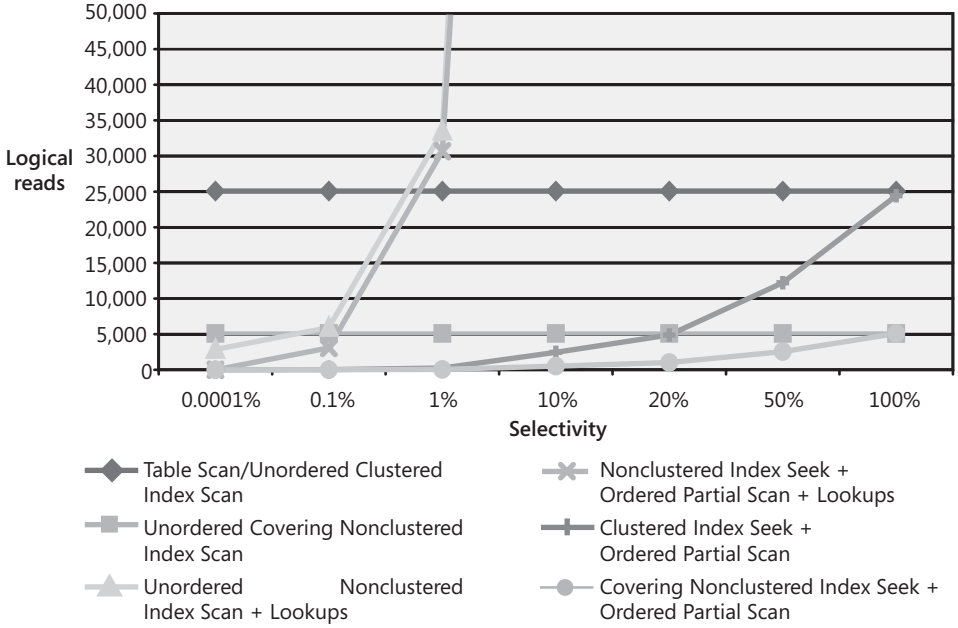
טבלה 16–3: קריאות לוגיות מול סלקטיביות לכל שיטת גישה

Access Method	*1*	*1,000*	*10,000*	*100,000*	*200,000*	*500,000*	*1,000,000*	*rows*
	0.0001%	*0.1%*	*1%*	*10%*	*20%*	*50%*	*100%*	*selectivity*
Table Scan/ Unordered Clustered Index Scan	25,080	25,080	25,080	25,080	25,080	25,080	25,080	
Unordered Covering Nonclustered Index Scan	5,095	5,095	5,095	5,095	5,095	5,095	5,095	
Unordered Nonclustered Index Scan + Lookups	2,855	5,923	33,486	309,111	615,361	1,534,111	3,065,442	
Nonclustered Index Seek + Ordered Partial Scan + Lookups	6	3,078	30,667	306,547	613,082	1,532,685	3,073,741	
Clustered Index Seek + Ordered Partial Scan	4	29	249	2,447	4,890	12,219	24,433	
Covering Nonclustered Index Seek + Ordered Partial Scan	4	9	54	512	1,021	2,546	5,087	

שים לב: כדי להשתמש בתוכנית עבודה מסוימת, במקרה בו ה–optimizer העדיף תוכנית אחרת יעילה יותר, הייתי צריך להשתמש ב– table hint כדי להכריח אותו להשתמש באינדקס הרלוונטי.

כמובן, קריאות לוגיות לא צריכות להיות האינדיקציה היחידה עליה אתה נשען. זכור שלדפוסי I/O שונים יש ביצועים שונים, ושקריאות פיסיות יקרות הרבה יותר מאשר קריאות לוגיות. אך כאשר אתה רואה הבדל משמעותי בקריאות לוגיות בין שתי אפשרויות, זו לרוב אינדיקציה טובה לתוכנית המהירה יותר. תרשים 3–55 מציג בצורה גרפית את המידע מטבלה 3–16.

תרשים 3–55: גרף קריאות לוגיות מול סלקטיביות



תוכל להבחין בהרבה דברים מעניינים כאשר אתה מנתח את הגרף. למשל, תוכל לראות בצורה ברורה אילו תוכניות תלויות בסלקטיביות ואילו לא. וכן תוכל לראות את נקודת הסלקטיביות בה תוכנית אחת הופכת להיות טובה יותר מאחרת.

באופן דומה, טבלה 3–17 מציגה סיכום של סטטיסטיקות הביצועים של עלות השאילתה מול סלקטיביות.

טבלה 3–17: עלות תת–עץ משוערת מול סלקטיביות לכל שיטת גישה

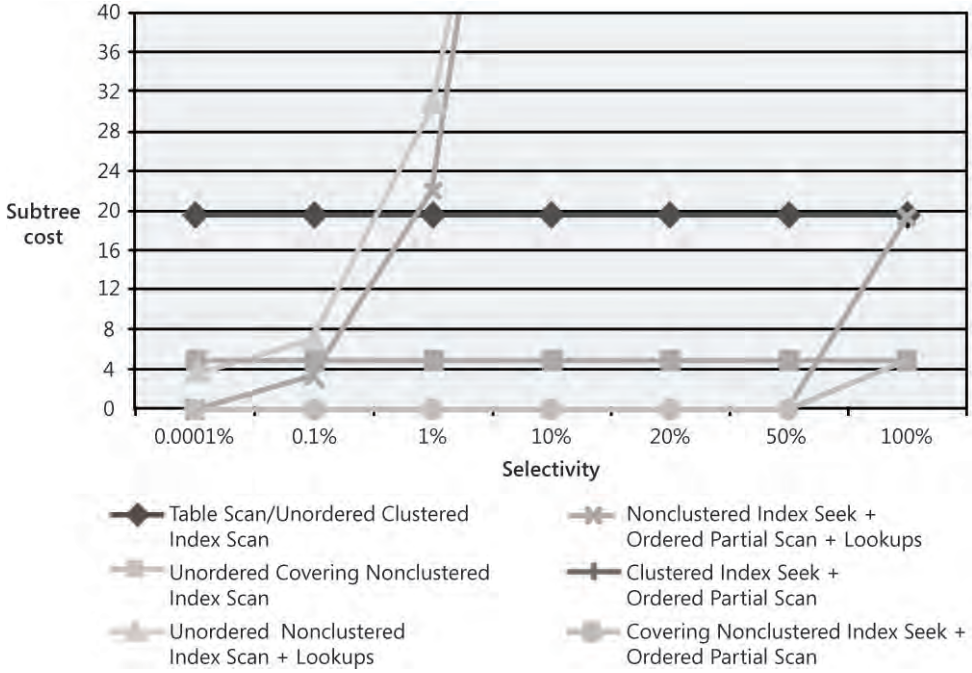
Access Method	*1*	*1,000*	*10,000*	*100,000*	*200,000*	*500,000*	*1,000,000*	*rows*
	0.0001%	*0.1%*	*1%*	*10%*	*20%*	*50%*	*100%*	*selectivity*
Table Scan/ Unordered Clustered Index Scan	19.621100	19.621100	19.621100	19.621100	19.621100	19.621100	19.621100	
Unordered Covering Nonclustered Index Scan	4.863280	4.863280	4.863280	4.863280	4.863280	4.863280	4.863280	
Unordered Nonclustered Index Scan + Lookups	3.690270	7.021360	30.665400	96.474000	113.966000	163.240000	244.092000	
Nonclustered Index Seek + Ordered Partial Scan + Lookups	0.006570	3.228520	21.921600	100.881000	127.376000	204.329000	335.109000	
Clustered Index Seek + Ordered Partial Scan	0.022160	0.022160	0.022160	0.022160	0.022160	0.022160	19.169900	
Covering Nonclustered Index Seek + Ordered Partial Scan	0.008086	0.008086	0.008086	0.008086	0.008086	0.008086	4.862540	

תרשים 3–56 מציג גרף המבוסס על הנתונים בטבלה 3–17.

תוכל להבחין בדמיון בולט בין שני הגרפים; אך אם תחשוב על כך, דבר זה הגיוני משום שרוב העלות המעורבת בתבנית השאילתה שלנו נגרמה בגלל I/O. באופן רגיל, בתוכניות בהן חלק משמעותי יותר של העלות קשורה למעבד, תקבל תוצאות אחרות.

כמובן, תרצה גם לייצר סטטיסטיקות וגרפים דומים עם זמני הריצה בפועל של השאילתות בבוחן הביצועים שלך. בסופו של יום, זמן ריצה הוא הדבר המשמעותי עבור המשתמש.

תרשים 56–3: גרף עלות תת–עץ לעומת סלקטיביות

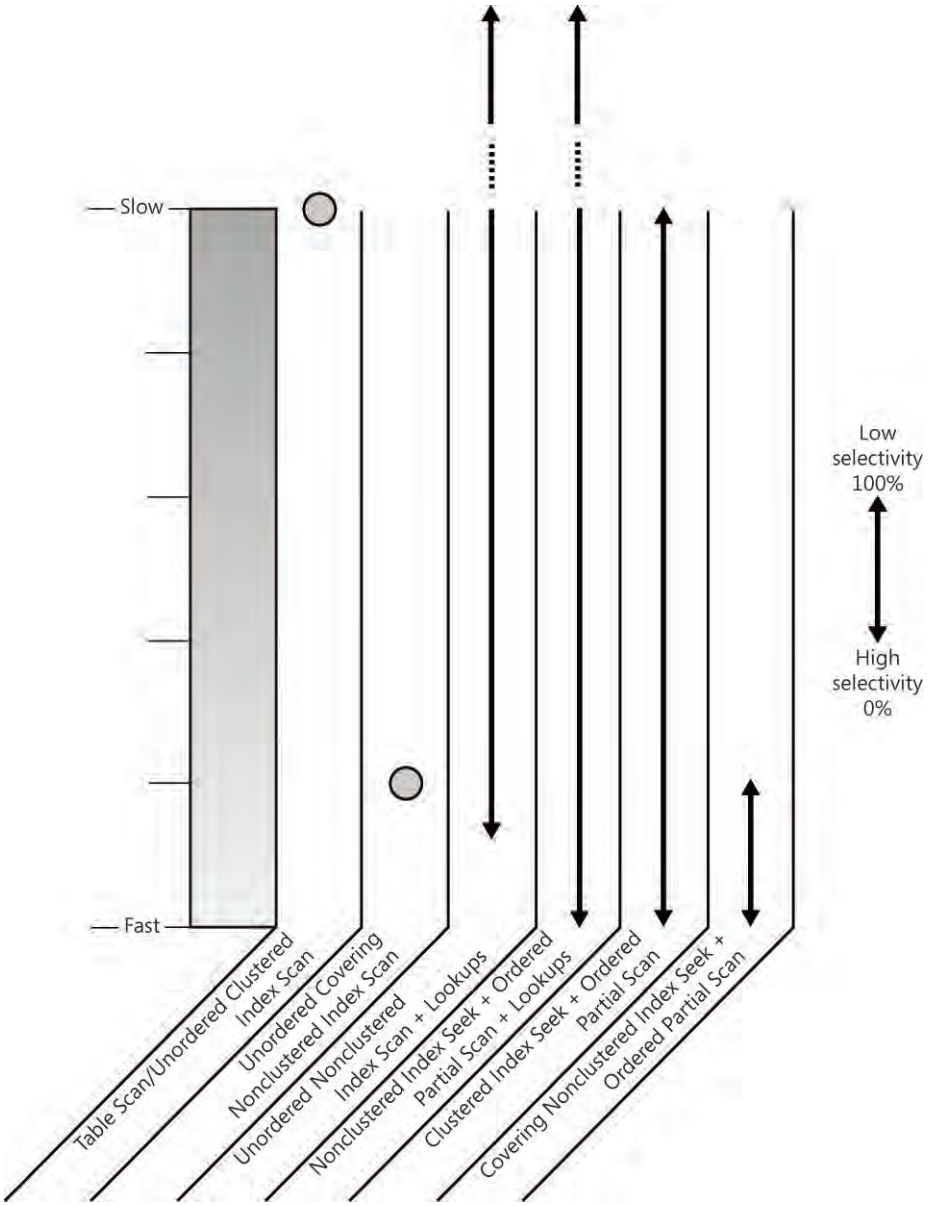


אני מוצא תועלת גם בבחינת מידע ביצועים בצורה גרפית אחרת כפי שמוצג בתרשים 57–3.

ייתכן ששרטוט זה יקל עליך לזהות תוכניות המבוססות על סלקטיביות לעומת תוכניות שלא (מוצגות כנקודה), וכן לערוך השוואות בין הביצועים של רמות אופטימיזציה שונות על הסרגל.

שים לב: לצורך הפשטות, כל הסטטיסטיקות והגרפים המוצגים בסעיף זה נאספו מול מסד הנתונים Performance בו השתמשתי בפרק זה, שבו רמת הפרגמנטציה של האינדקסים הייתה מינימלית. כאשר אתה עורך בחני ביצועים ובדיקות ביצועים, ודא שאתה מציג את הרמות הנכונות של פרגמנטציה באינדקסים במערכת הטסטים שלך כך שישקפו נאמנה את רמות הפרגמנטציה של האינדקסים במערכת התפעולית שלך. הביצועים של ordered scans עשויים להיות מושפעים בצורה משמעותית מרמת הפרגמנטציה של האינדקסים שלך. באופן דומה, עליך לבדוק גם את צפיפות הדף הממוצעת במערכת התפעולית שלך, ולהציג צפיפות דף דומה במערכת הבדיקות.

תרשים 57–3: סרגל אופטימיזציה של אינדקסים



מלבד היכולת לעצב אינדקסים טובים, חשוב גם לפתח את היכולת לזהות באילו אינדקסים משתמשים לעיתים יותר קרובות ואילו נמצאים בשימוש מועט או כלל לא. אינך רוצה לשמור אינדקסים שיש בהם שימוש לעיתים נדירות, שכן יש להם השפעה שלילית על הביצועים בעת שינויי נתונים. ב– SQL Server 2000, הטכניקות היחידות שהיו זמינות

לקביעת שימוש באינדקסים היו מאוד לא נוחות, אם נאמר בלשון המעטה. למשל, היית יכול לקחת קוד המכיל דוגמה מייצגת של השאילתות הרצות במערכת ולייצר מידע SHOWPLAN טקסטואלי עבור השאילתות בקוד. היית יכול לכתוב יישום המפרק את הפלט, מחלץ את שמות האינדקסים בהם משתמשים, ומחשב סטטיסטיקות שימוש באינדקסים. כמובן שזו טכניקה בעייתית הדורשת גם מאמץ רב למדי.

אפשרות נוספת בה יכולת להשתמש ב– SQL Server 2000 היא להריץ את ה– ITW (Index Tuning Wizard) בהתבסס על עומס עבודה לדוגמה. בין הדוחות ש–ITW מייצר יש דוח המראה סטטיסטיקות שימוש באינדקסים הקיימים עבור שאילתות הקלט.

ב– SQL Server 2005, יש לך כלים הרבה יותר חזקים ונוחים לעבודה. יש לך DMF הנקרא dm_db_index_operational_stats ו–DMV הנקרא dm_db_index_usage_stats. ה– DMF dm_db_index_operational_stats נותן לך מידע על I/O ברמות נמוכות, נעילות, latching ופעילות שיטות גישה. אתה מספק לפונקציה קוד מסד נתונים, קוד אובייקט, קוד אינדקס (או 0 עבור heap), וקוד מחיצה. באפשרותך לבקש גם מידע על מספר ישויות על ידי הגדרת NULL בארגומנט הרלוונטי. למשל, כדי לקבל מידע על כל האובייקטים, אינדקסים ומחיצות במסד הנתונים Performance, תקרא לפונקציה כלהלן:

```
SELECT *
FROM sys.dm_db_index_operational_stats(
  DB_ID('Performance'), null, null, null);
```

ה– DMV dm_db_index_usage_stats נותן לך סטטיסטיקות של שימוש בשיטות הגישה השונות:

```
SELECT *
FROM sys.dm_db_index_usage_stats;
```

אובייקטי הניהול הדינמיים הופכים את הניתוח של שימוש באינדקסים קל ומדויק הרבה יותר מאשר בעבר.

פרגמנטציה

בפרק זה התייחסתי במספר הזדמנויות לפרגמנטציה של אינדקסים. כאשר הזכרתי פרגמנטציה, התייחסתי לסוג הידוע כ– logical scan fragmentation או average fragmentation in percent או external fragmentation. כפי שציינתי קודם לכן, סוג זה משקף את האחוז של דפים מחוץ–לסדר באינדקס, במונחים של הסדר הפיסי שלהם לעומת הסדר הלוגי שלהם ברשימה המקושרת. זכור שלפרגמנטציה זו עשויה להיות השפעה משמעותית על פעולות ordered scan באינדקסים. אין לה כל השפעה על פעולות שאינן נשענות על הרשימה המקושרת של האינדקס – למשל, פעולות חיפוש,

lookups, unordered scans, וכו'. ברצונך להקטין את רמת הפרגמנטציה של אינדקסים עבור שאילתות שחלק משמעותי של עלותן קשורה ב– ordered scans. אתה עושה זאת על ידי בנייה–מחדש של אינדקסים.

סוג אחר של פרגמנטציה שלרוב יהיה חשוב לך הוא זה שאני מתייחס אליו כצפיפות ממוצעת של דף. ישנם מקצועני מסדי נתונים המתייחסים לסוג כזה של פרגמנטציה בשם internal fragmentation, אך לשם מניעת בלבול לא השתמשתי במונח זה קודם במודע. בעוד פרגמנטציה של סריקה לוגית אף פעם אינה דבר טוב, לפרגמנטציה של צפיפות ממוצעת יש שני היבטים. לאחוז נמוך (רמה נמוכה של מילוי דף) יש השפעה שלילית על שאילתות הקוראות נתונים, שכן הן קוראות בסופו של דבר יותר דפים משיכלו פוטנציאלית אם הדפים היו מלאים יותר. ההשפעה החיובית של מקום פנוי בדפי אינדקס היא שהוספת שורות בדפים כאלו לא תגרום לפיצול דפים–שהוא יקר מאוד. כפי שתוכל לנחש, מקום פנוי בדפי אינדקס גרוע במערכות המערבות בעיקר קריאות (למשל, data warehouses) וטוב במערכות המערבות בעיקר הוספת נתונים (למשל, מערכות OLTP). ייתכן שתרצה אפילו ליצור מקום פנוי בדפי אינדקס על ידי ציון ערך fillfactor כאשר אתה בונה–מחדש את האינדקסים שלך.

כדי לקבוע האם אתה צריך לבנות–מחדש או לארגן–מחדש את האינדקסים שלך, אתה זקוק למידע על שני סוגי הפרגמנטציה. ב– SQL Server 2005, תוכל להשיג מידע זה על ידי ביצוע שאילתה על ה– dm_db_index_physical_stats DMF. למשל, השאילתה הבאה תחזיר מידע פרגמנטציה על האינדקסים במסד הנתונים Performance:

```
SELECT *
FROM sys.dm_db_index_physical_stats(
  DB_ID('Performance'), NULL, NULL, NULL, NULL);
```

סוגי הפרגמנטציה שהזכרתי יופיעו בטורים avg_fragmentation_in_percent ו– avg_page_space_used_in_percent, וכפי שתוכל לראות, שמות הטורים מסבירים את עצמם. ב– SQL Server 2000, אתה משתמש בפקודה DBCC SHOWCONTIG לקבלת מידע דומה:

```
DBCC SHOWCONTIG WITH ALL_INDEXES, TABLERESULTS, NO_INFOMSGS;
```

שני הטורים המעניינים לצורך הדיון שלנו הם **LogicalFragmentation** ו– **AveragePageDensity**.

כפי שהזכרתי קודם, כדי לטפל בשני סוגי הפרגמנטציה עליך לבנות–מחדש או לארגן–מחדש את האינדקס שלך. לבנייה–מחדש של אינדקס יש את השפעת הדה–פרגמנטציה האופטימלית. הפעולה מבצעת את הניסיון הטוב ביותר שלה לבנות–מחדש את האינדקס באותו סדר פיסי על הדיסק כמו ברשימה המקושרת וליצור דפים רציפים ככל האפשר. כמו כן, זכור שבאפשרותך לציין fillfactor כדי ליצור מקום פנוי בדפי העלה של האינדקס.

ב– SQL Server 2000, בניות–מחדש של אינדקסים היו פעולות offline. בנייה–מחדש של אינדקס–clustered דרשה נעילה בלעדית לכל משך הפעולה, כלומר שתהליכים לא יכלו לקרוא מהטבלה או לכתוב אליה. בנייה–מחדש של אינדקס–nonclustered גרמה לנעילה משותפת, כלומר לא ניתן לבצע כתיבה לטבלה, וכמובן, לא ניתן להשתמש באינדקס במהלך הפעולה. SQL Server 2005 מציג פעולות אינדקס online המאפשרות לך ליצור, לבנות–מחדש ולהסיר אינדקסים online. בנוסף, פעולות אלו מאפשרות למשתמשים לתקשר עם הנתונים בזמן שהפעולה מתרחשת. פעולות אינדקס online משתמשות בטכנולוגית ה– row-versioning החדשה שהוצגה ב– SQL Server 2005. כאשר אינדקס נבנה–מחדש online, SQL Server למעשה מתחזק מאחורי הקלעים שני אינדקסים, וכאשר הפעולה מסתיימת, האינדקס החדש דורס את הישן.

לדוגמה, הקוד הבא בונה–מחדש online את האינדקס idx_cl_od על טבלת Orders:

```
ALTER INDEX idx_cl_od ON dbo.Orders REBUILD WITH (ONLINE = ON);
```

שים לב שפעולות אינדקס online זקוקות למספיק מקום במסד הנתונים והן בסך הכל איטיות יותר מאשר פעולות offline. אם באפשרותך לפנות חלון תחזוקה כדי להפעיל פעולות אלו offline, עדיף שתעשה כך. אפילו כאשר אתה בוחר לבצע את הפעולות online, יהיה להן השפעה על ביצועי המערכת כל עוד הן רצות, כך שעדיף להריץ אותן מחוץ לשעות השיא.

במקום לבנות–מחדש אינדקס (rebuild), באפשרותך גם לארגן אותו מחדש (defrag). ארגון–מחדש של אינדקס מיישם אלגוריתם מיון בועות למיון דפי האינדקס פיסית על הדיסק בהתאם לסדר שלהם ברשימה המקושרת של האינדקס. הפעולה לא מנסה לגרום לדפים להיות יותר סמוכים (לצמצם רווחים). כפי שתוכל לנחש, רמת הדה–פרגמנטציה שאתה מקבל מפעולה זו אינה אופטימלית כמו בנייה–מחדש של האינדקס. כמו כן, בסך הכל, פעולה זו מבצעת יותר רישום ללוג מאשר בנייה–מחדש של אינדקס ולפיכך, היא לרוב איטית יותר.

אם כך, מדוע להשתמש בסוג זה של דה–פרגמנטציה? ראשית, ב– SQL Server 2000 היה זה כלי הדה–פרגמנטציה ה–online היחיד. הפעולה משתמשת בנעילות לטווח–קצר על זוג דפים בכל פעם כדי לקבוע האם הם בסדר הנכון, ואם אינם, היא מחליפה ביניהם. שנית, בנייה–מחדש של אינדקס חייבת לרוץ כטרנזקציה יחידה, ואם היא מופסקת באמצע התהליך, כל הפעילות עוברת roll back. זאת שלא כמו פעולת ארגון–מחדש של אינדקס, שיכולה להיות מופרעת שכן היא פועלת על זוג דפים בכל פעם. כשאתה מריץ מאוחר יותר את פעולת הארגון–מחדש שוב, היא תמשיך מהנקודה בה הפסיקה.

כך תבצע ארגון–מחדש לאינדקס idx_cl_od ב– SQL Server 2005:

```
ALTER INDEX idx_cl_od ON dbo.Orders REORGANIZE;
```

ב– SQL Server 2000, אתה משתמש בפקודה DBCC INDEXDEFRAG לאותה מטרה.

חציצה (Partitioning)

SQL Server 2005 מציג חציצה מובנית (native partitioning) של טבלאות ואינדקסים. חציצה של האובייקטים שלך משמעותה שהם מפוצלים פנימית למספר יחידות פיסיות המרכיבות יחד את האובייקט (טבלה או אינדקס). חציצה היא למעשה בלתי נמנעת בסביבות בינוניות עד גדולות. על ידי חציצת האובייקטים שלך, אתה משפר את היכולת לנהל ולתחזק את המערכת שלך ואתה משפר ביצועים של פעולות כגון מחיקת נתונים היסטוריים, טעינת נתונים, ואחרות. חציצה ב– SQL Server היא מובנית – כלומר, יש לך כלים מובנים לחציצה של הטבלאות והאינדקסים, בעוד שלוגית, ליישום ולמשתמשים הם נראים יחידות שלמות. ב– SQL Server 2000, חציצה הושגה על ידי יצירה ידנית של מספר טבלאות ועליהן view המאחד את החלקים. לחציצה המובנית של SQL Server 2005 יש יתרונות רבים על פני partitioned views ב– SQL Server 2000, ביניהם שיפור תוכניות עבודה, רשימת אילוצים נמוכה בהרבה, ועוד.

מידע נוסף: חציצה היא מחוץ לטווח ספר זה, אך כיסיתי אותה בפירוט בסדרת מאמרים ב- SQL Server Magazine. תוכל למצוא מאמרים אלו בכתובת: http://www.sqlmag.com (חפש קודי מאמרים 45153, 45533, 45877 ו-46207).

הכנת נתונים לדוגמה

כאשר מבצעים בדיקות ביצועים, חיוני ביותר שהנתונים לדוגמה בהם אתה משתמש יוכנו בצורה נאותה כך שישקפו את המערכת התפעולית בצורה מדויקת ככל האפשר, במיוחד בהקשר לגורמים אותם אתה מנסה לכוונן. לרוב, אין זה ריאלי פשוט להעתיק את כל הנתונים מטבלאות המערכת התפעולית, לפחות לא את הגדולות. עם זאת, עליך לעשות מאמץ להכין ייצוג הולם, אשר ישקף פיזור דומה של נתונים, צפיפות מפתחות, קרדינאליות, וכו'. כמו כן, אתה רוצה שלשאילתות שלך על מערכת הבדיקות תהיה סלקטיביות דומה לזו של השאילתות על המערכת התפעולית. בדיקות ביצועים עלולות להיות מעוותות כאשר הנתונים לדוגמה אינם מייצגים בצורה הולמת את הנתונים התפעוליים.

בסעיף זה, אספק דוגמה של תוצאות בדיקות ביצועים מעוותות הנובעות מנתוני דוגמה בלתי הולמים. אדון גם באפשרות TABLESAMPLE החדשה.

הכנת נתונים

כאשר הכנתי את נתוני הדוגמה להדגמות של פרק זה, לא היה עלי לשקף מערכת תפעולית מסוימת, כך שהכנת הנתונים לדוגמה הייתה פשוטה למדי. הזדקקתי לנתונים בעיקר בסעיפים "מתודולוגית כוונון" ו"כוונון אינדקסים". יכולתי לבטא את מרבית הנקודות שלי דרך פיזור רנדומאלי פשוט של ערכים בטורים השונים שהיו רלוונטיים לדיון שלנו. אך טבלת הנתונים העיקרית שלנו – Orders – לא משקפת בצורה מדויקת טבלת Orders תפעולית ממוצעת. למשל, ייצרתי פיזור ערכים אחיד למדי בטורים השונים,

בעוד שבמערכות תפעוליות טיפוסיות, לטורים שונים יש סוגי פיזור שונים (חלקם אחידים, חלקם סטנדרטיים). ישנם לקוחות המבצעים הזמנות רבות, ואחרים מבצעים מעטות. כמו כן, ישנם לקוחות הפעילים יותר בתקופות זמן מסוימות ופעילים פחות באחרות. בהתאם לצרכי הכוונון שלך, תצטרך או שלא לשקף דברים כאלו בנתוני הדוגמה שלך, אך ללא ספק עליך לקחת אותם בחשבון ולהחליט אם הם אכן חשובים.

כאשר אתה נזקק לטבלאות גדולות עם נתונים לדוגמה, הדבר הקל ביותר לביצוע הוא לייצר טבלה קטנה כלשהי ולהכפיל את התוכן שלה (מלבד טורי המפתח) פעמים רבות. דבר זה יכול להיות בסדר אם, למשל, ברצונך לבדוק את הביצועים של פונקציה מוגדרת–משתמש המופעלת על כל שורה, או מניפולציה של סמן בלולאה על שורות רבות, שורה בכל איטרציה. אך נתוני דוגמה כאלה במקרים מסוימים עשויים להניב ביצועים שונים לחלוטין מאשר אלו שתקבל עם נתוני דוגמה המשקפים בצורה נאותה יותר את הנתונים התפעוליים שלך. כדי להדגים זאת, אלווה אותך דרך דוגמה אותה אני מכסה בפירוט רב הרבה יותר בספר **תכנות T-SQL**. לעיתים קרובות אני נותן את התרגיל הזה בכיתה ומבקש מתלמידים להכין כמות גדולה של נתוני דוגמה מבלי לתת שום רמזים.

התרגיל עובד על טבלה הנקראת Sessions, אותה אתה יכול ליצור ולמלא בנתונים על ידי הרצת הקוד בקטע–קוד 6–3.

קטע קוד 6–3: יצירה ומילוי של טבלת Sessions

```
SET NOCOUNT ON;
USE Performance;
GO
IF OBJECT_ID('dbo.Sessions') IS NOT NULL
  DROP TABLE dbo.Sessions;
GO

CREATE TABLE dbo.Sessions
(
  keycol    INT             NOT NULL IDENTITY,
  app       VARCHAR(10)     NOT NULL,
  usr       VARCHAR(10)     NOT NULL,
  host      VARCHAR(10)     NOT NULL,
  starttime DATETIME NOT NULL,
  endtime   DATETIME NOT NULL,
  CONSTRAINT PK_Sessions PRIMARY KEY(keycol),
  CHECK(endtime > starttime)
);
```

```
INSERT INTO dbo.Sessions
  VALUES('app1', 'user1', 'host1', '20030212 08:30', '20030212 10:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user2', 'host1', '20030212 08:30', '20030212 08:45');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user3', 'host2', '20030212 09:00', '20030212 09:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user4', 'host2', '20030212 09:15', '20030212 10:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user5', 'host3', '20030212 09:15', '20030212 09:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user6', 'host3', '20030212 10:30', '20030212 14:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user7', 'host4', '20030212 10:45', '20030212 11:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user8', 'host4', '20030212 11:00', '20030212 12:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user8', 'host1', '20030212 08:30', '20030212 08:45');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user7', 'host1', '20030212 09:00', '20030212 09:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user6', 'host2', '20030212 11:45', '20030212 12:00');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user5', 'host2', '20030212 12:30', '20030212 14:00');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user4', 'host3', '20030212 12:45', '20030212 13:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user3', 'host3', '20030212 13:00', '20030212 14:00');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user2', 'host4', '20030212 14:00', '20030212 16:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user1', 'host4', '20030212 15:30', '20030212 17:00');

CREATE INDEX idx_nc_app_st_et ON dbo.Sessions(app, starttime, endtime);
```

טבלת Sessions מכילה מידע על Sessions של משתמשים מול יישומים שונים. הבקשה היא לחשב את המספר המקסימלי של sessions שהיו פעילים בו–זמנית לכל יישום.

השאילתה הבאה מייצרת את המידע המבוקש, כפי שמוצג בטבלה 3–18:

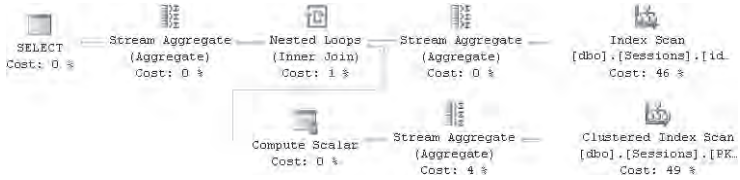
```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.Sessions AS S2
         WHERE S1.app = S2.app
           AND S1.ts >= S2.starttime
           AND S1.ts < S2.endtime) AS concurrent
      FROM (SELECT DISTINCT app, starttime AS ts
            FROM dbo.Sessions) AS S1) AS C
GROUP BY app;
```

טבלה 3–18: מקסימום Sessions פעילים בו–זמנית לכל יישום

app	*mx*
app1	4
app2	3

הטבלה הנגזרת S1 מכילה זוגות של שם היישום הייחודי (app) וזמן התחלה של ה– session (starttime as ts). לכל שורה של S1, סופרת תת–שאילתה את מספר ה–sessions שהיו פעילים לאפליקציה S1.app בזמן S1.ts. השאילתה החיצונית אז מקבצת את הנתונים לפי app ומחזירה את המספר המקסימלי בכל קבוצה. ה–optimizer של SQL Server מפיק עבור השאילתה הזו את תוכנית העבודה המוצגת בתרשים 3–58.

תרשים 3–58: תוכנית עבודה לשאילתה מול טבלת Sessions



בקוד בקטע–קוד 3–6, ייצרתי את האינדקס–המכסה idx_nc_app_st_et על (app, starttime, endtime), שהוא האינדקס האופטימלי לשאילתה זו. בתוכנית, אינדקס זה נסרק לפי הסדר (אופרטור Index Scan) ושורות ייחודיות מבודדות (אופרטור Stream Aggregate). כאשר שורות זורמות החוצה מהאופרטור Stream Aggregate, אופרטור Nested Loops קורא לסדרת פעילויות לחישוב מספר ה–sessions הפעילים לכל שורה. משום שטבלת Sessions כה

קטנה (רק דף אחד של נתונים), ה–optimizer פשוט מחליט לסרוק את הטבלה כולה (unordered clustered index scan) לחישוב כל ספירה. עם סט נתונים גדול יותר, במקום לסרוק את הטבלה, התוכנית הייתה מבצעת seek ו– ordered partial scan של האינדקס–המכסה לקבלת כל ספירה. לבסוף, אופרטור Stream Aggregate אחר מקבץ את הנתונים לפי app לחישוב הספירה המקסימלית לכל קבוצה.

כעת כשאתה מבין את הבעיה, נניח שהיית מתבקש להכין נתוני דוגמה עם 1,000,000 שורות בטבלת המקור (נקרא לה BigSessions) כך שהיא תייצג סביבה מציאותית. אידיאלית, היית צריך לחשוב על פיזור נתוני ה–sessions על פני תקופה (חודש, במקרה שלנו). אלא שאנשים נוקטים לעיתים קרובות בגישה הכי מובנת מאליה, שהיא להכפיל את הנתונים מטבלת המקור הקטנה פעמים רבות; במקרה שלנו, גישה כזו תעוות את הביצועים משמעותית בהשוואה לייצוג מציאותי יותר של סביבות תפעוליות.

כעת הרץ את הקוד בקטע–קוד 7–3 כדי לייצר את הטבלה BigSessions על ידי הכפלת הנתונים מטבלת Sessions פעמים רבות. תקבל 1,000,000 שורות בטבלה BigSessions.

קטע–קוד 7–3: מילוי BigSessions בנתוני דוגמה בלתי הולמים

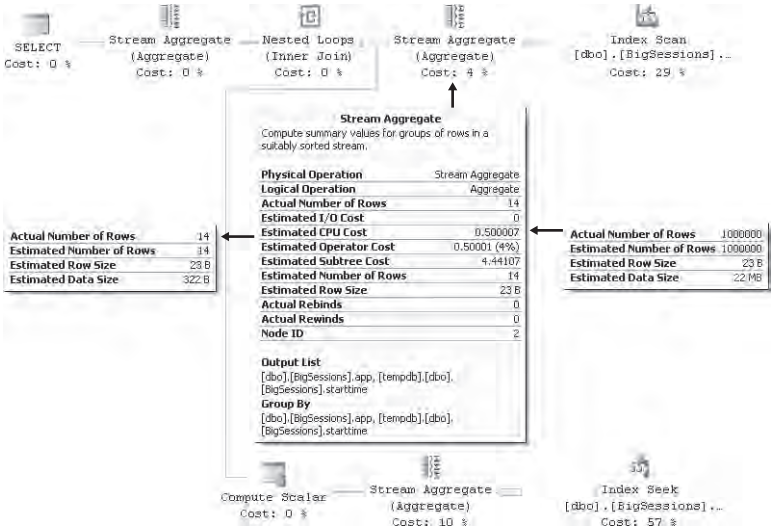
```
SET NOCOUNT ON;
USE Performance;
GO
IF OBJECT_ID('dbo.BigSessions') IS NOT NULL
  DROP TABLE dbo.BigSessions;
GO
SELECT IDENTITY(int, 1, 1) AS keycol,
  app, usr, host, starttime, endtime
INTO dbo.BigSessions
FROM dbo.Sessions AS S, Nums
WHERE n <= 62500;

ALTER TABLE dbo.BigSessions
  ADD CONSTRAINT PK_BigSessions PRIMARY KEY(keycol);
CREATE INDEX idx_nc_app_st_et
  ON dbo.BigSessions(app, starttime, endtime);
```

הרץ את השאילתה הבאה על BigSessions:

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.BigSessions AS S2
         WHERE S1.app = S2.app
           AND S1.ts >= S2.starttime
           AND S1.ts < S2.endtime) AS concurrent
     FROM (SELECT DISTINCT app, starttime AS ts
           FROM dbo.BigSessions) AS S1) AS C
GROUP BY app;
```

שים לב שזוהי אותה שאילתה כמקודם (אלא שמול טבלה שונה). השאילתה תסיים לאחר מספר שניות, ותקבל את תוכנית העבודה המוצגת בתרשים 59–3.

תרשים 59–3: תוכנית עבודה על טבלת BigSessions עם נתוני דוגמה בלתי הולמים



להלן מדידות הביצועים שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 20024
- זמן מעבד: 1452 ms
- זמן שעון: 1531 ms
- עלות תת–עץ משוערת: 13.6987

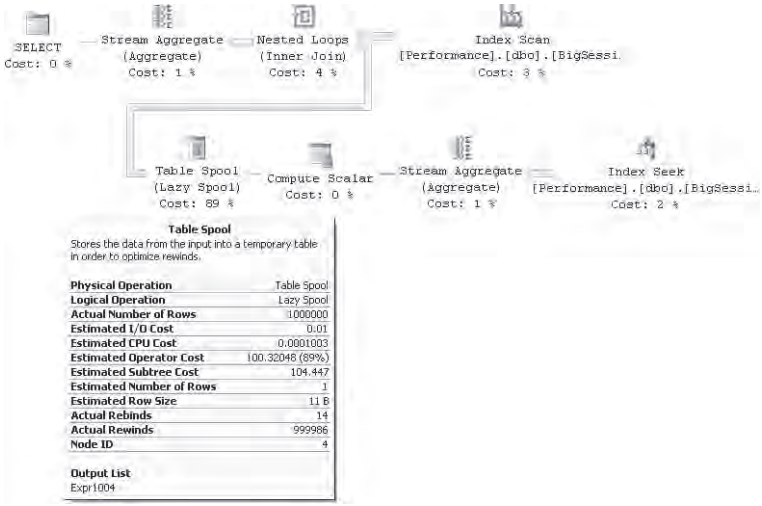
כמובן, לאחר שהאופרטור Stream Aggregate הראשון העלים שורות משוכפלות, הוא הניב 14 שורות ייחודיות; כך שהאופרטור Nested Loops שבא אחריו הפעיל רק 14 איטרציות של הפעילות המחשבת את ספירת ה–sessions הפעילים. לפיכך, השאילתה סיימה בתוך מספר שניות בלבד. בסביבה תפעולית, עשויים להיות רק יישומים בודדים, אך לא סביר שיהיו כל–כך מעט זמני תחילת עבודה ייחודיים.

ייתכן שתחשוב שמובן מאליו שנתוני הדוגמה אינם הולמים משום שהצעד הראשון בשאילתה היוצר את הטבלה הנגזרת S1 מבודד ערכי app ו–starttime ייחודיים – כלומר הצבירה COUNT בתת–השאילתה תופעל רק מספר קטן של פעמים. אך במקרים רבים, העובדה שיש עיוות בביצועים כתוצאה מנתוני דוגמה גרועים אינה מובנת מאליה. למשל, תוכניתנים רבים יחשבו על השאילתה הבאה כפתרון לבעיה שלנו, בה הם לא יבודדו קודם כל ערכי app ו–starttime ייחודיים:

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.BigSessions AS S2
         WHERE S1.app = S2.app
           AND S1.starttime >= S2.starttime
           AND S1.starttime < S2.endtime) AS concurrent
      FROM dbo.BigSessions AS S1) AS C
GROUP BY app;
```

בחן את תוכנית העבודה שנוצרה עבור שאילתה זו, כפי שמוצגת בתרשים 60–3.

תרשים 60–3: תוכנית עבודה לשאילתה על טבלת BigSessions עם נתוני דוגמה בלתי הולמים



התמקד באופרטור Table Spool, המייצג טבלה זמנית המחזיקה את ספירת ה–sessions לכל שילוב ייחודי של ערכי app ו–starttime. משום שלא בודדת ערכי app ו–starttime ייחודיים, ה–optimizer עושה זאת בצורה אלגנטית עבורך. שים לב למספר ה–rebinds (14) ולמספר ה–rewinds (999,986). זכור שמשמעותו של rebind היא שפרמטר מקושר אחד או יותר של אופרטור ה–join השתנה והצד הפנימי חייב להיות מוערך מחדש. דבר זה קרה 14 פעמים, פעם אחת לכל זוג ייחודי של app ו–starttime – כלומר, פעילות הספירה בפועל שלפני האופרטור קרתה רק 14 פעמים. משמעותו של rewind היא שאף אחד מהפרמטרים המקושרים לא השתנה והתוצאה הפנימית הקודמת ניתנת לשימוש חוזר; דבר זה קרה 999,986 פעמים (1,000,000-14=999,986). באופן טבעי, עם פיזור נתונים מציאותי יותר לתרחיש שלנו, פעילות הספירה תקרה הרבה יותר פעמים מאשר 14, ותקבל שאילתה איטית הרבה יותר. הייתה זאת טעות להכין את נתוני הדוגמה על ידי העתקה פשוטה של השורות מטבלת Sessions הקטנה פעמים רבות. פיזור הנתונים בטורים השונים צריך לייצג סביבות תפעוליות בצורה מציאותית יותר.

הרץ את הקוד בקטע–קוד 8–3 למילוי BigSessions בנתוני דוגמה הולמים יותר.

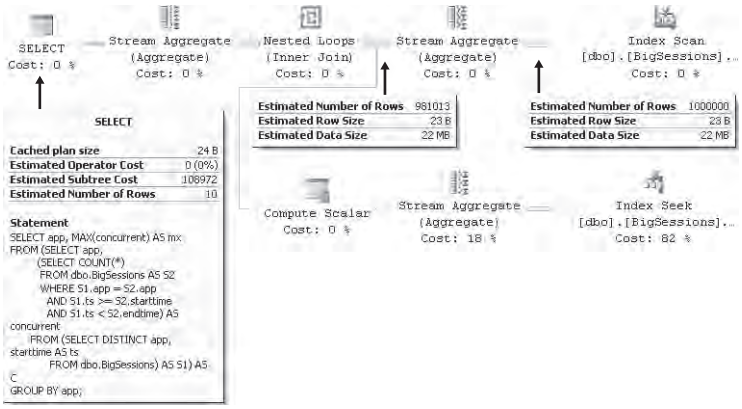
קטע–קוד 8–3: מילוי טבלת BigSessions בנתוני דוגמה הולמים

```
SET NOCOUNT ON;
USE Performance;
GO
IF OBJECT_ID('dbo.BigSessions') IS NOT NULL
  DROP TABLE dbo.BigSessions;
GO

SELECT
  IDENTITY(int, 1, 1) AS keycol,
  D.*,
  DATEADD(
    second,
    1 + ABS(CHECKSUM(NEWID())) % (20*60),
    starttime) AS endtime
INTO dbo.BigSessions
FROM
(
  SELECT
    'app' + CAST(1 + ABS(CHECKSUM(NEWID())) % 10 AS VARCHAR(10)) AS app,
    'user1' AS usr,
    'host1' AS host,
```

```
    DATEADD(
      second,
      1 + ABS(CHECKSUM(NEWID())) % (30*24*60*60),
      '20040101') AS starttime
  FROM dbo.Nums
  WHERE n <= 1000000
) AS D;

ALTER TABLE dbo.BigSessions
  ADD CONSTRAINT PK_BigSessions PRIMARY KEY(keycol);
CREATE INDEX idx_nc_app_st_et
  ON dbo.BigSessions(app, starttime, endtime);
```

מילאתי את הטבלה ב–sessions המתחילים בזמנים רנדומאליים על פני תקופה של חודש אחד ואשר מתמשכים עד 20 דקות. פיזרתי גם 10 שמות יישומים שונים בצורה רנדומאלית. כעת בקש תוכנית עבודה משוערת לשאילתה המקורית ותקבל את התוכנית המוצגת בתרשים 61–3.

תרשים 61–3: תוכנית עבודה משוערת לשאילתה על טבלת BigSessions עם נתוני טבלה הולמים.



עלות השאילתה כעת היא 108,972. סמוך עלי; אתה לא רוצה להריץ אותה כדי לראות כמה זמן היא באמת רצה. או, אם אתה רוצה, אתה יכול להתחיל להריץ אותה ולחזור למחרת בתקווה שסיימה.

כעת כשהנתונים לדוגמה מציאותיים יותר, תוכל לראות שהפתרון מבוסס–הסטים שהוצג בסעיף זה איטי – שלא כמו מה שהיית חושב לאחר שימוש בנתוני דוגמה בלתי הולמים. בקיצור, אתה יכול לראות עד כמה חיוני להשקיע מחשבה בהכנת נתוני דוגמה טובים.

כמובן, בזאת תהליך הכוונון רק מתחיל; ייתכן שתרצה לשקול שינויים לשאילתה, פתרונות מבוססי–סמן, שינויים למודל הנתונים, וכו'. אך כאן רציתי למקד את הדיון בנתוני דוגמה גרועים. בספר **תכנות T-SQL** אנהל דיון כוונון מעמיק יותר הקשור לבעיה זו.

TABLESAMPLE

SQL Server 2005 מציג מאפיין חדש המאפשר לך לדגום נתונים מטבלה קיימת. הכלי הוא פסוקית הנקראת TABLESAMPLE אותה אתה מגדיר לאחר שם הטבלה בפסוקית FROM בצירוף מספר אפשרויות. להלן דוגמה לשימוש ב–TABLESAMPLE כדי לבקש 1000 שורות מטבלת Orders במסד הנתונים Performance:

```
SELECT *
FROM Performance.dbo.Orders TABLESAMPLE (1000 ROWS);
```

שים לב שאם תריץ שאילתה זו סביר להניח שלא תקבל בדיוק 1000 שורות, מייד אסביר מדוע.

תוכל לציין TABLESAMPLE עבור כל טבלה. לאחר מילת המפתח TABLESAMPLE, באפשרותך להגדיר באיזו שיטת דגימה להשתמש. נכון לעכשיו, SQL Server תומך רק בשיטה SYSTEM, שהיא גם ברירת המחדל אם לא צוינה כל שיטה. בעתיד, ייתכן שנראה אלגוריתמים נוספים. לפי ANSI, מילת המפתח SYSTEM מייצגת שיטת דגימה תלוית–הטמעה. המשמעות היא שתמצא אלגוריתמים שונים מוטמעים במוצרים שונים כאשר תשתמש בשיטה SYSTEM. ב– SQL Server, השיטה SYSTEM מטמיעה את אותו אלגוריתם דגימה המשמש לדגימת דפים לייצור סטטיסטיקות.

תוכל להשתמש באחת ממילות המפתח ROWS או PERCENT כדי להגדיר כמה שורות תרצה לקבל חזרה. בהתבסס על הקלטים שלך, SQL Server יחשב ערכים רנדומאליים כדי לדעת האם דף צריך להיות מוחזר או שלא. שים לב שההחלטה האם לקרוא חלק מהנתונים או לא מתבצעת ברמת הדף. עובדה זו, בנוסף לדרך בה SQL Server קובע האם לקחת דף או שלא בהתבסס על גורם רנדומאלי, משמעותה שלא בהכרח תקבל את מספר השורות המדויק אותו ביקשת; אלא, תקבל ערך קרוב למדי. ככל שתבקש יותר שורות, כך תקבל גודל סט תוצאה קרוב למספר שביקשת.

להלן דוגמה לשימוש בפסוקית TABLESAMPLE בשאילתה על טבלת Orders, המבקשת 1,000 שורות:

```
SET NOCOUNT ON;
USE Performance;

SELECT *
FROM dbo.Orders TABLESAMPLE SYSTEM (1000 ROWS);
```

הרצתי שאילתה זו שלוש פעמים וקיבלתי מספר שורות שונה בכל פעם: 880, 1200 ו–920.

יתרון חשוב שאתה מקבל בשיטת הדגימה SYSTEM הוא שרק הדפים הנבחרים (אלו ש– SQL Server בחר) ייסרקו פיסית. כך שאפילו אם אתה מבצע שאילתה על טבלה ענקית, תקבל את התוצאות מהר למדי – כל עוד אתה מגדיר מספר שורות קטן למדי. כפי שציינתי קודם, באפשרותך גם לציין אחוז שורות. להלן דוגמה לבקשת 0.1 אחוז, השווה ל–1000 שורות בטבלה שלנו:

```
SELECT *
FROM dbo.Orders TABLESAMPLE (0.1 PERCENT);
```

כאשר אתה משתמש באפשרות ROWS, SQL Server ראשית ממיר פנימית את מספר השורות המצוין לאחוז. זכור שלא מובטח לך שתקבל את מספר השורות המדויק שביקשת; אלא, תקבל ערך קרוב הנקבע על ידי מספר הדפים שנבחרו ומספר השורות בדפים אלו (שעשוי להיות שונה מדף לדף).

כדי להגדיל את הסבירות שתקבל את מספר השורות בו אתה מעוניין, ציין מספר גדול יותר של שורות בפסוקית TABLESAMPLE והשתמש באפשרות TOP כדי להגביל את הגבול העליון שתקבל, כלהלן:

```
SELECT TOP(1000) *
FROM dbo.Orders TABLESAMPLE (2000 ROWS);
```

עדיין יש סיכוי שתקבל פחות שורות מאשר המספר שביקשת אך מובטח לך שלא תקבל יותר. על ידי הגדרת ערך גבוה יותר בפסוקית TABLESAMPLE, אתה מגדיל את הסבירות לקבלת מספר השורות בו אתה מעוניין.

אם אתה נדרש לקבל תוצאות ניתנות–לחזרה, השתמש בפסוקית הנקראת REPEATABLE שעוצבה למטרה זו, וספק לה אותו זרע (seed) בכל קריאה. למשל, הרצת השאילתה הבאה מספר פעמים תניב את אותה תוצאה, בהנחה שהנתונים בטבלה לא השתנו:

```
SELECT *
FROM dbo.Orders TABLESAMPLE (1000 ROWS) REPEATABLE(42);
```

שים לב שעם טבלאות קטנות ייתכן שלא תקבל שורות כלל. למשל, הרץ מספר פעמים את השאילתה הבאה, המבקשת שורה יחידה מטבלת Sales.StoreContact במסד הנתונים AdventureWorks:

```
SELECT *
FROM AdventureWorks.Sales.StoreContact TABLESAMPLE (1 ROWS);
```

רק מדי פעם תקבל שורות בחזרה. נתקלתי בדיון מאוד מעניין בפורום טכני של SQL Server. מישהו הציג שאילתה כזו ורצה לדעת מדוע הוא לא קיבל אף שורה בחזרה. סטיב קאס, ידיד שלי והעורך הטכני המבריק של ספרים אלו, סיפק את התשובה הבאה המבהירה את העניין והרשה לי לצטט אותה כאן:

"כפי שמתועד ב– Books Online ("Limiting Results Sets by Using TABLESAMPLE"), אלגוריתם הדגימה יכול להחזיר רק דפים מלאים. כל דף נבחר או מדולג בסבירות של [מספר מבוקש של שורות] חלקי [שורות בטבלה].

הטבלה StoreContact נכנסת ב–4 דפי נתונים. שלושה מדפים אלו מכילים 179 שורות, ואחד מכיל 37 שורות. כאשר אתה דוגם עבור 10 שורות (1/75 מהטבלה), כל אחד מ–4 הדפים מוחזר בסבירות של 1/75 ומדולג בסבירות של 74/75. הסיכוי שאף שורה לא תוחזר היא בערך 4^(74/75), או בערך 87%. כאשר שורות מוחזרות, בערך ב–3/4 מהזמן תראה 179 שורות, ובערך ב–1/4 מהזמן תראה 37 שורות. לעיתים מאוד נדירות, תראה יותר שורות, אם שני דפים או יותר מוחזרים, אך הסבירות לכך מאוד נמוכה.

כפי ש–BOL מציעים, דגימת SYSTEM (שהיא האפשרות היחידה) אינה מומלצת לטבלאות קטנות. אני הייתי מוסיף שאם הטבלה נכנסת ב–N דפי נתונים, אל לך לנסות לדגום פחות מ–1/N של השורות, או שעליך לעולם לא לנסות לדגום פחות שורות מאשר הכמות שנכנסת בלפחות 2 או 3 דפי נתונים.

אם היית דוגם כמות שורות השווה בערך לשני דפי נתונים, נאמר 300 שורות, הסיכוי לא לראות כלל שורות יהיה בערך 13%. ככל שהטבלה יותר גדולה (יותר דפי נתונים), כך קטן הסיכוי לקבלת אפס שורות כאשר מבוקשת לפחות כמות השווה לשני דפים. למשל, אם אתה מבקש 300 שורות מטבלה בת 1,000,000 שורות הנכנסת ב–10,000 דפי נתונים, רק ב–5% מהניסיונות תקבל אפס שורות, אפילו שהבקשה היא להרבה פחות מאשר 1% של השורות.

על ידי בחירה באפשרות REPEATABLE, תקבל את אותה דגימה בכל פעם. עבור מרבית הזרעים, היא תכיל 37, 179, 216, 358, 395, 537, 574, או 753 שורות, תלוי אילו דפים נבחרו, כאשר המספר הגדול יותר של שורות מוחזר עבור מעט מאוד בחירות של זרע.

עם זאת, אני מסכים שההשלכות של החזרת דפי נתונים מלאים בלבד מביאות להתנהגות מאוד מבלבלת!"

עם טבלאות קטנות, ייתכן שתרצה לשקול שיטות דגימה אחרות. אינך מוטרד יותר מדי מסריקה של הטבלה כולה משום שתשקול שיטות אלו מול טבלאות קטנות בכל מקרה. למשל, השאילתה הבאה תסרוק את הטבלה כולה, אך היא תבטיח שתקבל שורה רנדומאלית יחידה:

```
SELECT TOP(1) *
FROM AdventureWorks.Sales.StoreContact
ORDER BY NEWID();
```

שים לב שפלטפורמות מסדי נתונים אחרות, כמו DB2, מיישמות אלגוריתמים נוספים – למשל, אלגוריתם הדגימה של ברנולי. באפשרותך ליישם אותו ב– SQL Server על ידי שימוש בשאילתה הבאה, שסופקה על ידי סטיב קאס:

```
SELECT *
FROM AdventureWorks.Sales.StoreContact
WHERE ABS((customerid%customerid)+CHECKSUM(NEWID()))/POWER(2.,31) < 0.01
GO
```

הקבוע 0.01 הוא ההסתברות הרצויה (במקרה הזה, 1%) לבחירת שורה. הביטוי customerid%customerid נכלל כדי לגרום לפסוקית WHERE להיות מקושרת ולכפות את ההערכה שלה על כל שורה של StoreContact. בלעדיו, הערך של תנאי ה–WHERE יחושב רק פעם אחת, ואו שהטבלה כולה תוחזר או שלא יוחזרו שורות כלל. שים לב ששיטה זו תדרוש סריקה מלאה של הטבלה ועשויה לארוך זמן לא מועט בטבלאות גדולות. תוכל לבדוק אותה מול טבלת Orders שלנו ולהיווכח בעצמך.

בחינה של גישות מבוססות–סטים לעומת גישות איטרטיביות/פרוצדורליות, ותרגיל כוונון

עד כה בפרק, התמקדתי בעיקר בכוונון אינדקסים לשאילתות נתונות. עם זאת, בחלקו הגדול, כוונון שאילתות כולל שינויי קוד. כלומר, עם שאילתות שונות או קוד T-SQL שונה לעיתים אתה יכול לקבל תוכניות שונות משמעותית, עם עלויות וזמני ריצה שונים לחלוטין. בעולם מושלם, ה–optimizer האידיאלי יבין תמיד בדיוק מה אתה מנסה להשיג; ולכל צורה של שאילתה או קוד T-SQL המנסים להשיג את אותו הדבר, תקבל את אותה תוכנית – ורק את התוכנית הטובה ביותר כמובן. חבל, אך אנחנו עדיין לא שם. קיימים עדיין שיפורי ביצועים רבים שניתן להשיג על ידי שינוי הדרך בה אתה כותב את הקוד שלך. דבר זה יודגם בצורה מעמיקה לאורך ספרים אלו. כאן, אציג תהליך כוונון טיפוסי מלווה בדוגמה המתבסס על שינויים בקוד.

שים לב ששאילתות מבוססות–סטים לרוב עדיפות על פני פתרונות המבוססים על לוגיקה איטרטיבית/פרוצדורלית – כגון אלו המשתמשים בסמנים, לולאות וכד'. מלבד העובדה שפתרונות מבוססי–סטים דורשים לרוב הרבה פחות קוד, הם גם לרוב מערבים פחות תקורה מאשר סמנים. ישנה תקורה רבה הנובעת ממניפולציית רשומה–אחר–רשומה של סמנים. תוכל לבצע בחני ביצועים פשוטים כדי לבחון את הבדלי הביצועים. הרץ שאילתה שפשוט בוחרת את כל השורות מטבלה גדולה (בחר באפשרות המונעת יצירת הפלט בכלי הגרפי כדי שהזמן שייקח להציג את הפלט לא יילקח בחשבון). הרץ גם קוד סמן שפשוט סורק את כל שורות הטבלה רשומה–בכל–פעם. גם אם תשתמש בסמן הזמין המהיר ביותר – FAST_FORWARD (קדימה בלבד, קריאה בלבד) – תמצא שהשאילתה

מבוססת–הסטים רצה עשרות פעמים מהר יותר. מלבד התקורה הקשורה לסמן, קיימת גם סוגיה עם תוכנית העבודה. כאשר משתמשים בסמן, מפעילים גישה פיסית קשיחה מאוד לגישה לנתונים, משום שהקוד שלך מתמקד מאוד בשאלה כיצד להשיג את הנתונים. שאילתה מבוססת–סטים, מצד שני, מתמקדת ב–מה ברצונך להשיג ולא ב–איך להשיג זאת. טיפוסית, שאילתות מבוססות–סטים משאירות ל–optimizer הרבה יותר מקום לתמרון ודרך זמינה לבצע את מה שהוא טוב בו–אופטימיזציה.

זהו כלל האצבע. אף על פי כן, אני לרוב נזהר מאוד באימוץ כללי אצבע, במיוחד בהקשר של כוונון שאילתות – משום שאופטימיזציה היא עולם כה דינמי, ותמיד קיימים יוצאי דופן. למעשה, בכל הקשור לכוונון שאילתות, כלל האצבע העיקרי שלי הוא להיזהר מאימוץ כללי אצבע.

אתה תתקל במקרים בהם מאוד קשה להביס קוד סמן, ועליך להיות מסוגל לזהות אותם; אך מקרים אלו הם המיעוט. אדון בנושא זה באריכות בפרק 3, "Cursors" בספר **תכנות T-SQL**.

כדי להדגים תהליך כוונון המתבסס על שינויי קוד, אשתמש בטבלאות Orders ו–Shippers שלנו. הבקשה היא להחזיר מובילים שהיו פעילים בתקופה כלשהי, אך לא ביצעו שום פעילות מאז 1 בינואר 2001. כלומר, מוביל שעונה על הבקשה הוא כזה שעבורו אינך יכול למצוא הזמנה בתאריך 1 בינואר 2001 או אחריו. אינך מעוניין במובילים שלא ביצעו הזמנות כלל.

בטרם תתחיל לעבוד, הסר את כל האינדקסים מטבלת Orders, וודא שמוגדר לך רק האינדקס–clustered על הטור orderdate והמפתח הראשי (nonclustered) מוגדר על הטור orderid.

אם אתה מריץ שוב את הקוד מקטע–קוד 1–3, ודא שעבור הטבלה Orders, אתה משאיר רק את ההגדרות הבאות של האינדקס והמפתח הראשי:

```
SET NOCOUNT ON;
USE Performance;
CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);
ALTER TABLE dbo.Orders ADD
  CONSTRAINT PK_Orders PRIMARY KEY NONCLUSTERED(orderid);
```

כעת, הרץ את הקוד הבא להוספת מספר מובילים לטבלת Shippers ומספר הזמנות לטבלת Orders:

```
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('B', 'Shipper_B');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('D', 'Shipper_D');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('F', 'Shipper_F');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('H', 'Shipper_H');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('X', 'Shipper_X');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('Y', 'Shipper_Y');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('Z', 'Shipper_Z');
```

```
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000001, 'C0000000001', 1, 'B', '20000101');
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000002, 'C0000000001', 1, 'D', '20000101');
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000003, 'C0000000001', 1, 'F', '20000101');
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000004, 'C0000000001', 1, 'H', '20000101');
```

אתה אמור לקבל את קודי המובילים B, D, F ו–H בתוצאה. אלו המובילים היחידים שהיו פעילים בנקודה כלשהי, אך לא מאז 1 בינואר 2001.

במונחים של כוונון אינדקסים, לעיתים קשה לדעת מה האינדקסים האופטימליים מבלי שיש לך שאילתה קיימת. אך במקרה שלנו, כוונון אינדקסים הוא פשוט למדי ואפשרי מבלי שיש לך את קוד הפתרון מלכתחילה. כמובן, תרצה לחפש ערך orderdate מקסימלי לכל shipperid, כך שטבעי שהאינדקס האופטימלי יהיה אינדקס–מכסה–nonclustered המוגדר עם shipperid ו–orderdate כטורי המפתח, בסדר הבא:

```
CREATE NONCLUSTERED INDEX idx_nc_sid_od
  ON dbo.Orders(shipperid, orderdate);
```

אני מציע שבנקודה זו תנסה לחשוב על הפתרון בעל הביצועים הטובים ביותר שאתה יכול להשיג; אז השווה אותו עם הפתרונות שאציג.

כפתרון הראשון, אתחיל בקוד מבוסס–הסמן המוצג בקטע–קוד 9–3.

קטע–קוד 9–3: פתרון סמן

```
DECLARE
  @sid     AS VARCHAR(5),
  @od      AS DATETIME,
  @prevsid AS VARCHAR(5),
  @prevod  AS DATETIME;

DECLARE ShipOrdersCursor CURSOR FAST_FORWARD FOR
  SELECT shipperid, orderdate
  FROM dbo.Orders
  ORDER BY shipperid, orderdate;

OPEN ShipOrdersCursor;

FETCH NEXT FROM ShipOrdersCursor INTO @sid, @od;
```

```
SELECT @prevsid = @sid, @prevod = @od;

WHILE @@fetch_status = 0
BEGIN
  IF @prevsid <> @sid AND @prevod < '20010101' PRINT @prevsid;
  SELECT @prevsid = @sid, @prevod = @od;
  FETCH NEXT FROM ShipOrdersCursor INTO @sid, @od;
END

IF @prevod < '20010101' PRINT @prevsid;

CLOSE ShipOrdersCursor;

DEALLOCATE ShipOrdersCursor;
```

קוד זה מיישם אלגוריתם צבירת–נתונים פשוט למדי המבוסס על מיון. הסמן מוגדר על שאילתה הממיינת את הנתונים לפי shipperid ו–orderdate, והיא סורקת את הרשומות בצורה קדימה–בלבד, קריאה–בלבד–הסריקה המהירה ביותר שניתן להשיג עם סמן. לכל מוביל, הקוד בוחן את השורה האחרונה שנמצאה – שהיא זו המחזיקה את ה–orderdate המקסימלי עבור מוביל זה – ואם תאריך זה מוקדם מ–'20010101', הקוד מחזיר את ערך ה–shipperid. הקוד רץ על המכונה שלי במשך 27 שניות. תאר לך את זמן הריצה בטבלת Orders גדולה יותר המכילה מיליוני שורות.

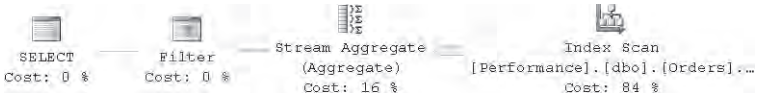
הפתרון הבא (נקרא לו פתרון מבוסס–סטים 1) הוא שאילתת GROUP BY טבעית, בה יבחרו תוכניתנים רבים:

```
SELECT shipperid
FROM dbo.Orders
GROUP BY shipperid
HAVING MAX(orderdate) < '20010101';
```

אתה פשוט אומר מה אתה רוצה, במקום לבזבז את מרבית הקוד שלך בתיאור כיצד להשיג זאת. השאילתה מקבצת את הנתונים לפי shipperid ומחזירה רק מובילים עם orderdate מקסימלי המוקדם מ–'20010101'.

השאילתה רצה במשך מעט פחות משנייה אחת במכונה שלי. ה–optimizer הפיק את תוכנית העבודה המוצגת בתרשים 3–62.

תרשים 62–3: תוכנית עבודה לפתרון מבוסס–סטים 1



התוכנית מראה שהאינדקס–המכסה שלנו נסרק במלואו לפי הסדר. ה–orderdate המקסימלי בודד לכל shipperid על ידי האופרטור Stream Aggregate. אז אופרטור הסינון סינן רק מובילים עבורם ה–orderdate המקסימלי היה לפני '20010101'.

להלן מדידות הביצועים החיוניות שקיבלתי עבור שאילתה זו:

- קריאות לוגיות: 2730
- זמן מעבד: 670 ms
- זמן שעון: 732 ms

שים לב שייתכן כי תקבל מדידות ביצועים מעט שונות. בנקודה זו, עליך לשאול את עצמך אם אתה מרוצה מהתוצאה, ואם אינך מרוצה, האם קיים כלל פוטנציאל לאופטימיזציה.

כמובן, פתרון זה הוא שיפור משמעותי יחסית לזה מבוסס–הסמן, הן במונחים של ביצועים והן במונחים של קריאות וקלות–תחזוקה של קוד. עם זאת, זמן ריצה של קרוב לשנייה אחת עבור שאילתה כזו עשוי שלא להיות משביע רצון. זכור שישנן סביבות תפעוליות בהן טבלת Orders יכולה להכיל הרבה יותר מאשר מיליון שורות.

אם אתה מחליט שברצונך לכוונן עוד את הפתרון, כעת עליך לזהות האם קיים פוטנציאל לאופטימיזציה. זכור שבתוכנית העבודה עבור השאילתה האחרונה, רמת העלה של האינדקס נסרקה בצורה מלאה כדי לקבל את ה–orderdate המאוחר ביותר עבור כל מוביל. סריקה זו דרשה 2730 קריאות דף. טבלת Shippers שלנו מכילה 12 מובילים. תחושת הבטן שלך צריכה לומר לך שחייבת להיות דרך להשיג את הנתונים בעזרת הרבה פחות קריאות. באינדקס שלנו, השורות ממוינות לפי shipperid ו–orderdate. המשמעות היא שישנן קבוצות של שורות – קבוצה לכל shuipperid – בהן השורה האחרונה בכל קבוצה מכילה את ה–orderdate המאוחר ביותר שברצונך לבחון. חבל, ל–optimizer נכון לעכשיו, אין את הלוגיקה הנדרשת לבצע "זיגזג" בין רמות האינדקס, כשהוא קופץ מ–orderdate מאוחר ביותר של מוביל אחד לזה של השני. אם היה לו, השאילתה הייתה מניבה כתוצאה מכך הרבה פחות I/O. דרך אגב, לוגיקת זיגזג כזו יכולה להוות יתרון גם עבור סוגי בקשות אחרות–למשל, בקשות המערבות מסננים על טור אינדקס שאינו ראשון ואחרות. אך לא אסטה כעת מהנושא.

כמובן, אם אתה מבקש את ה–orderdate המאוחר ביותר עבור מוביל מסוים, ה–optimizer יכול להשתמש ב–seek ישירות לשורה האחרונה של המוביל באינדקס. Seek כזה יעלה 3 קריאות במקרה שלנו. אז ה–optimizer יכול להפעיל אופרטור TOP כשהוא הולך צעד אחד אחורה, ומחזיר את הערך המבוקש – ה–orderdate האחרון עבור המוביל הנתון לאופרטור Stream Aggregate.

השאילתה הבאה מדגימה השגת ה–orderdate המאוחר ביותר למוביל מסוים, ומפיקה את תוכנית העבודה המוצגת בתרשים 3–63:

```
SELECT MAX(orderdate) FROM dbo.Orders WHERE shipperid = 'A';
```

תרשים 3–63: תוכנית עבודה עבור שאילתה המטפלת במוביל מסוים

תוכנית זו עולה רק 5 קריאות לוגיות. כעת, אם תעשה את החשבון עבור 12 מובילים, תראה שאתה יכול פוטנציאלית לקבל את התוצאה הרצויה עם הרבה פחות I/O מאשר 2730 קריאות. כמובן, תוכל לסרוק את שורות Shippers עם סמן, ואז לקרוא לשאילתה כזו לכל מוביל; אך יהיה זה בלתי יעיל ומעט אירוני להביס פתרון סמן עם פתרון מבוסס–סטים, שאותו אתה מייד מביס עם סמן אחר.

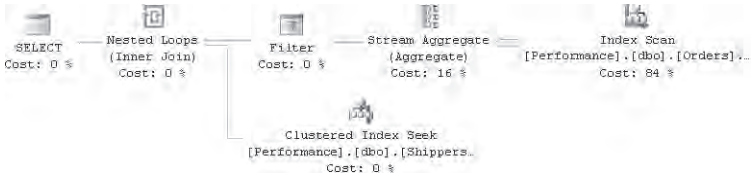
כשאתה מבין שמה שאתה צריך הוא קריאה לפעולת seek לכל מוביל, ייתכן שתעשה את הניסיון הבא (קרא לו פתרון מבוסס–סטים 2):

```
SELECT   shipperid
FROM    (SELECT shipperid,
           (SELECT MAX(orderdate)
            FROM dbo.Orders AS O
            WHERE O.shipperid = S.shipperid) AS maxod
         FROM dbo.Shippers AS S) AS D
WHERE maxod < '20010101';
```

אתה מבצע שאילתה על טבלת Shippers, ולכל מוביל, תת–שאילתה מוצאת את ערך ה–orderdate המאוחר ביותר (המקבל כינוי maxod). השאילתה החיצונית אז מסננת רק מובילים עם ערך maxod מוקדם יותר מאשר '20010101'. מובילים אשר מעולם לא ביצעו הזמנה יסוננו החוצה; לאלו, תת–השאילתה תניב NULL, ו–NULL בהשוואה לכל ערך יניב UNKNOWN, שבתורו יסונן החוצה.

אך באופן מוזר, אתה מקבל את התוכנית המוצגת בתרשים 3–64, הנראית באופן מפתיע דומה לקודמת.

תרשים 3–64: תוכנית עבודה לפתרון מבוסס–סטים 2



ומדידות הביצועים גם הן דומות לאלו הקודמות. שאילתה זו גם היא מביאה ל–2730 קריאות לוגיות על טבלת Orders, ורצה קרוב לשנייה אחת על המכונה שלי. אז מה קרה פה?

נראה שה–optimizer היה "יותר מדי מתוחכם" הפעם. ניחוש אחד הוא שהוא הבין שעבור מובילים שלא ביצעו הזמנות כלל, תת–השאילתה תחזיר NULL והם יסוננו החוצה במסנן של השאילתה החיצונית–כלומר שמעניינים רק מובילים שכן מופיעים בטבלת Orders. לפיכך, הוא "פישט" את השאילתה לכזו הדומה לפתרון מבוסס–הסטים 1 שלנו, ומכאן התוכניות הדומות.

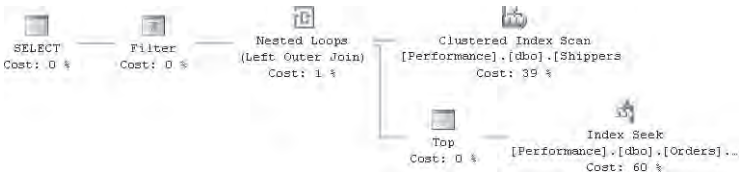
נראה שהמצב מתפתח לכדי מאבק מוחות עם ה–optimizer. ה–optimizer מבצע תרגיל עליך, כעת שלוף אתה את התרגיל הטוב ביותר שלך. אתה יכול לגרום ל–optimizer להאמין שמובילים ללא הזמנות עשויים להיות מעניינים גם הם, על ידי החלפת ערך NULL המוחזר מתת–השאילתה בקבוע; תוכל לעשות זאת על ידי שימוש בפונקציה COALESCE. כמובן, תשתמש בקבוע שלעולם לא יאפשר לביטוי המסנן להניב TRUE, שכן אינך מעוניין במובילים שאין להם הזמנות.

אתה קורא לשאילתה הבאה (קרא לה פתרון מבוסס–סטים 3), עוצם את עיניך, ומקווה לטוב:

```
SELECT shipperid
FROM  (SELECT shipperid,
         (SELECT MAX(orderdate)
          FROM dbo.Orders AS O
          WHERE O.shipperid = S.shipperid) AS maxod
       FROM dbo.Shippers AS S) AS D
WHERE COALESCE(maxod, '99991231') < '20010101';
```

וכאשר אתה פותח את עיניך, אתה מוצא את התוכנית לה קיווית, כפי שמוצג בתרשים 3–65.

תרשים 3–65: תוכנית עבודה לפתרון מבוסס–סטים 3



טבלת Shippers נסרקת, ולכל אחד מ–12 המובילים, אופרטור Nested Loops קורא לפעילות דומה לזו שקיבלת כשקראת לשאילתה עבור מוביל מסוים. תוכנית זו מביאה ל–60 קריאות לוגיות בלבד. זמן המעבד נטו אינו ניתן אפילו למדידה עם STATISTICS TIME (מופיע כ–0), ואני קיבלתי 26 אלפיות שנייה של זמן שעבר מתחילת הריצה.

ברגע שאתה נרגע מההתרגשות על כך שהתחכמת ל–optimizer, אתה מתחיל לגלות כמה עובדות מטרידות. שאילתה כזו אינה טבעית כל–כך, ואינה אינטואיטיבית ביותר עבור תוכניתנים להבין. היא משתמשת בטכניקה מאוד מלאכותית, במטרה יחידה של התגברות על סוגיית optimizer – דבר הדומה ל–hint, אך עם הפעלת לוגיקה ערמומית. תוכניתנים אחרים שיצטרכו לתחזק את הקוד שלך בעתיד עשויים לבהות בקוד במשך זמן מה ולשאול, "מה לעזאזל חשב לעצמו התוכניתן שכתב את השאילתה הזו? אני יכול לעשות זאת בצורה הרבה יותר פשוטה". התוכניתן אז ישנה את הקוד חזרה לפתרון מבוסס–סטים 1. כמובן, תוכל להוסיף הערות המתעדות את הטכניקה ואת ההיגיון שמאחוריה. אך הומור בצד, הנקודה היא ששאילתות צריכות להיות פשוטות, טבעיות, ואינטואיטיביות, כך שאנשים יוכלו לקרוא ולהבין אותן – כמובן, עד כמה שהדבר אפשרי. פתרון מורכב, מהיר ככל שיהיה, אינו פתרון טוב. זכור שהמקרה שלנו הוא תרחיש פשוט המשמש למטרות הדגמה. שאילתות תפעוליות טיפוסיות מערבות מספר joins, יותר מסננים, ויותר לוגיקה. הוספת "טריקים" כאלו תוסיף רק למורכבות של השאילתות.

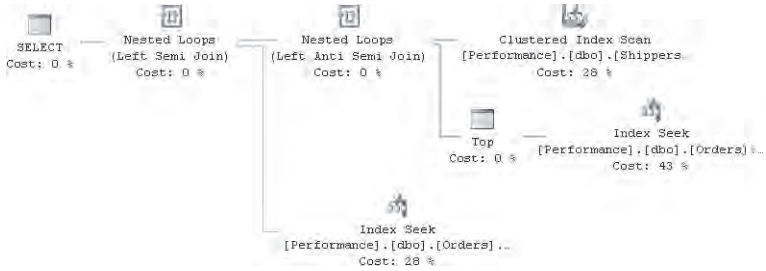
בקיצור, עליך להיות לא מסופק מפתרון זה, ועליך להמשיך לחפש אלטרנטיבות אחרות. ואם תחפש מספיק, תמצא את זו (קרא לה פתרון מבוסס–סטים 4):

```
SELECT shipperid
FROM dbo.Shippers AS S
WHERE NOT EXISTS
  (SELECT * FROM dbo.Orders AS O
   WHERE O.shipperid = S.shipperid
     AND O.orderdate >= '20010101')
  AND EXISTS
  (SELECT * FROM dbo.Orders AS O
   WHERE O.shipperid = S.shipperid);
```

פתרון זה הוא טבעי. אתה מבצע שאילתה על טבלת Shippers ומסנן מובילים עבורם אינך יכול למצוא הזמנה בתאריך '20010101' או אחריו ואשר עבורם אתה יכול למצוא לפחות הזמנה אחת.

תקבל את התוכנית המוצגת בתרשים 66–3.

תרשים 66–3: תוכנית עבודה לפתרון מבוסס–סטים 4



טבלת Shippers נסרקת, ומניבה 12 שורות. לכל מוביל, אופרטור Nested Loops קורא ל–seek על האינדקס–המכסה שלנו כדי לבדוק האם orderdate של '20010101' או מאוחר יותר קיים עבור המוביל. אם התשובה היא כן, פעולת seek נוספת נקראת על האינדקס כדי לבדוק האם בכלל קיימת הזמנה. עלות ה–I/O מול טבלת Orders היא 78 קריאות–מעט יותר גבוהה מאשר בפתרון הקודם. עם זאת, במונחים של פשטות וטבעיות פתרון זה מנצח בגדול! לפיכך, הייתי דבק בו.

כפי שודאי ניחשת, כוונון אינדקסים לבדו אינו מספיק; יש הרבה מה לעשות עם הדרך בה אתה כותב את השאילתות שלך. כחובב "Matrix", אני נוטה להאמין ש– "It's not the spoon that bends; it's only your mind".

מקורות נוספים

כפי שהבטחתי, להלן רשימת מקורות בהם תוכל למצוא מידע נוסף בנוגע לנושאים שנידונו בפרק זה:

- בהמשך ספר זה, תמצא כיסוי של טכניקות שאילתות בהן השתמשתי בפרק זה, ביניהן תת–שאילתות, ביטויי טבלה ופונקציות דירוג, joins, צבירות, TOP, שינויי נתונים, שאילתות רקורסיביות ואחרות.
- הספר Inside Microsoft SQL Server 2005: T-SQL Programming מתמקד בתחומים הקשורים לתכנות ב–T-SQL, כמה מהם הוזכרו בפרק זה, ביניהם טיפוסי נתונים, טבלאות זמניות, tempdb, row versioning, סמנים, הפעלה דינמית, רוטינות T-SQL ו–CLR, קומפילציות/קומפילציות–מחדש, עבודה של ריבוי משתמשים בו–זמנית, ואחרים.
- הספרים Inside Microsoft SQL Server 2005: The Storage Engine ו–Inside Microsoft SQL Server 2005: Query Tuning and Optimization, שניהם נכתבו על ידי Kalen Delaney (Microsoft Press, 2006 ו–2007), מכסים בצורה מעמיקה נושאים שונים הקשורים ל–internals, ביניהם ה– storage engine, אינדקסים, ארכיטקטורה של לוג הטרנזקציות, query optimizer, עבודה של ריבוי משתמשים בו–זמנית, אובייקטי ניהול דינמיים, tracing ו–Profiler, ואחרים.
- תוכל למצוא מידע נוסף לגבי סוגי המתנות וניתוח סטטיסטיקות המתנות בקישורים הבאים:
  - Opening Microsoft's Performance-Tuning Toolbox / Tom Davidson (http://www.windowsitpro.com/Article/ArticleID/40925/40925.html)
  - האתר של Gert Drapers: SQLDev.Net (http://www.sqldev.nrt)
- תוכל למצוא מידע על עצים מאוזנים במקורות הבאים:
  - http://www.nist.gov/dads/HTML/balancedtree.html

- The Art of Computer Programming, Volume 3: Sorting and Searching (מהדורה שנייה) מאת Donald E. Knuth (Addison-Wesley Professional, 1998)

◉ המאמרים הבאים מרחיבים את הדיבור על נושאים שנדונו בפרק זה:

- "Troubleshooting Performance Problems in SQL Server 2005" מאת כותבים שונים (http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx).
- "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005" מאת Arun Marathe (http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx).
- "Statistics Used by the Query Optimizer in Microsoft SQL Server 2005" מאת Eric N. Hanson (http://www.microsoft.com/technet/prodtechnol/sql/2005/qrystats.mspx).
- "Forcing Query Plans with SQL Server 2005" מאת Burzin A. Patel (http://download.microsoft.com/download/1/3/4/134644FD-05AD-4EE8-8B5A-0AED1C18A31E/Forcing_Query_Plans.doc).
- "Improving Performance with SQL Server 2005 Indexed Views" מאת Eric Hansom (http://www.microsoft.com/technet/prodtechnol/sql/2005/ipsql05iv.mspx).
  "Database Concurrency and Row Level Versioning in SQL Server 2005" מאת Kalen Delaney and Fernando G. Guerrero (http://www.microsoft.com/technet/prodtechnol/sql/2005/cncrrncy.mspx).

◉ תוכל למצוא סדרת מאמרים שכתבתי המכסים חציצה של טבלאות ואינדקסים באתר של SQL Server Magazine (http://www.sqlmag.com). חפש את קודי המאמרים הבאים: 45153, 45533, 45877 ו–46207.

◉ תוכל למצוא את כל המקורות הקשורים לספר זה באתר http://www.insidetsql.com.

סיכום

פרק זה כיסה מתודולוגיית כוונון, כוונון אינדקסים, החשיבות של נתוני דוגמה, וכוונון שאילתות על ידי שינויי קוד. בכוונון מעורבים דברים כה רבים; ידע על הארכיטקטורה ועל ה–internals של המוצר משחק תפקיד חשוב בביצוע כוונון טוב. אך ידע אינו מספיק. אני מקווה שפרק זה נתן לך את הכלים ואת ההדרכה שתאפשר לך להפעיל ידע זה ככל שאתה מתקדם בספרים אלו – וכמובן, בסביבות התפעוליות שלך.

4

תת–שאילתות, ביטויי טבלה ופונקציות דירוג

פרק זה ביסודו מכסה קינון של שאילתות. בתוך שאילתה חיצונית ניתן לכלול שאילתה פנימית (ידועה גם כתת–שאילתה). ניתן להשתמש בתת–שאילתות כאשר צפוי ערך יחידני (תת–שאילתות סקלאריות) – למשל, לימינו של סימן שווה בביטוי לוגי. ניתן להשתמש בהן כאשר צפויים ערכים מרובים (תת–שאילתות מרובות–ערכים) – למשל, כקלט לאופרטור IN. ניתן גם להשתמש בהן כאשר צפויה טבלה (ביטויי טבלה) – למשל, בפסוקית FROM בשאילתה.

בהמשך אתייחס לתת–שאילתות סקלאריות ומרובות–ערכים כ**תת–שאילתות**, ולתת–שאילתות בהן משתמשים כאשר צפויה טבלה כ**ביטויי טבלה**. בפרק זה אכסה ביטויי טבלה פנימיים (inline): טבלאות נגזרות וביטויי טבלה שגורים (CTE).

בסעיף האחרון של הפרק אכסה **חישובי דירוג**: ביניהם מספר–שורה, rank, dense rank וריצוף (tiling). מצאתי לנכון לכסות חישובי דירוג בפרק זה מכיוון שב– Microsoft SQL Server 2000 ניתן היה להשתמש בתת–שאילתות לצורך חישובים אלו. SQL Server 2005 מציג פונקציות דירוג מובנות, המאפשרות לחשב ערכים אלו בצורה פשוטה ויעילה הרבה יותר.

מכיוון שספר זה מיועד למפתחים מנוסים, אני מניח כי אתה כבר מכיר תת–שאילתות וביטויי טבלה. אני אגדיר אותם בקצרה, ואתמקד ביישומים שלהם ובפתרון בעיות.

תת–שאילתות

תת–שאילתות ניתנות לאפיון בשתי דרכים עיקריות. אחת היא לפי מספר הערכים הצפוי (סקלארית או מרובת–ערכים), והשנייה היא לפי התלות של תת–השאילתה בשאילתה החיצונית (עצמאית או תלויה). הן תת–שאילתה סקלארית והן תת–שאילתה מרובת–ערכים יכולות להיות עצמאיות או תלויות.

תת–שאילתות עצמאיות

תת–שאילתה עצמאית (Self-Contained Subqueries) היא תת–שאילתה שניתן להריץ בנפרד מהשאילתה החיצונית. בתת–שאילתות עצמאיות נוח מאוד לאתר באגים, כמובן, יחסית לתת–שאילתות מקושרות.

תת–שאילתות סקלאריות יכולות להופיע בכל מקום בשאילתה בו צפוי ביטוי שתוצאתו היא ערך יחידני, בעוד שתת–שאילתות מרובות–ערכים יכולות להופיע בכל מקום בשאילתה שבו צפוי אוסף של ערכים מרובים.

תת–שאילתה סקלארית הינה תקינה כאשר היא מחזירה ערך יחיד וכן כאשר היא אינה מחזירה ערכים כלל – במקרה כזה הערך של תת–השאילתה הוא NULL. במידה שתת–שאילתה סקלארית מחזירה יותר מערך אחד, תופיע שגיאה בזמן ריצה.

לדוגמה, הרץ את הקוד הבא שלוש פעמים: פעם אחת כפי שמוצג, פעם שנייה רשום LIKE N'kollar' במקום LIKE N'Davolio', ופעם שלישית רשום LIKE N'D%'.

```
SET NOCOUNT ON;
USE Northwind;

SELECT OrderID
FROM dbo.Orders
WHERE EmployeeID =
  (SELECT EmployeeID FROM dbo.Employees
  -- also try with N'Kollar' and N'D%' in place of N'Davolio'
  WHERE LastName LIKE N'Davolio');
```

עם N'Davolio', תת–השאילתה מחזירה ערך אחד (1) והשאילתה החיצונית מחזירה את כל ההזמנות שעבורן EmployeeID הוא 1.

עם N'Kollar', תת–השאילתה אינה מחזירה ערכים ולכן תוצאתה היא NULL. השאילתה החיצונית אינה מוצאת כמובן אף הזמנה שעבורה EmployeeID = NULL ולכן מחזירה סט ריק. שים לב שהשאילתה אינה נכשלת, שכן זו שאילתה תקנית.

עם N'D%', תת–השאילתה מחזירה שני ערכים (1, 9), ומכיוון שהשאילתה החיצונית מצפה לערך יחידני, הריצה נכשלת ומציגה את הודעת השגיאה הבאה:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the
subquery follows =, !=, <, <= , >, >= or when the subquery is used as
an expression.
```

לוגית, תת–שאילתה עצמאית יכולה להיות מוערכת פעם אחת בלבד עבור כל שאילתה חיצונית. פיסית, ה–optimizer יכול לנקוט בהרבה דרכים שונות להשיג את אותה מטרה, כך שרצוי לא לחשוב במונחים כה קשיחים.

לאחר שעברנו על עקרונות הבסיס, הבה נמשיך לבעיות מתוחכמות יותר המערבות תת–שאילתות עצמאיות.

אתחיל בבעיה השייכת לקבוצת בעיות הנקראת **חלוקה רלציונית** (Relational Division). לבעיות של חלוקה רלציונית קיימים ניואנסים ויישומים מעשיים רבים. לוגית, זה כמו חלוקה של סט אחד באחר, להפקת סט תוצאה. למשל, ממסד הנתונים Northwind החזר את כל הלקוחות שעבורם כל העובדים מארה"ב טיפלו לפחות בהזמנה אחת. במקרה זה אתה מחלק את הסט של כל ההזמנות בסט של כל העובדים מארה"ב, ואתה מצפה לקבל בחזרה את הסט של הלקוחות המתאימים. סינון כאן אינו פשוט כל–כך שכן עבור כל לקוח עליך לבחון מספר שורות כדי לדעת אם קיבלת התאמה.

בשלב זה אציג שיטה המשתמשת ב– GROUP BY וב– DISTINCT COUNT כדי לפתור בעיות של חלוקה רלציונית. בהמשך הספר אציג שיטות נוספות.

אם הייתה ידועה לך מראש רשימת כל ה– EmployeeID עבור עובדים מארה"ב, היית יכול לכתוב את השאילתה הבאה כדי לפתור את הבעיה, והיית מקבל את הפלט המוצג בטבלה 1–4:

```
SELECT CustomerID
FROM dbo.Orders
WHERE EmployeeID IN(1, 2, 3, 4, 8)
GROUP BY CustomerID
HAVING COUNT(DISTINCT EmployeeID) = 5;
```

טבלה 1–4: לקוחות עם הזמנות שטופלו על ידי כל העובדים מארה"ב

CustomerID
BERGS
BONAP
ERNSH
FOLKO
HANAR
HILAA
HUNGO
ISLAT
KOENE
LAMAI
LILAS

CustomerID
LINOD
LONEP
MEREP
OLDWO
QUICK
RATTC
SAVEA
TORTU
VAFFE
VICTE
WARTH
WHITC

שאילתה זו מוצאת את כל ההזמנות עם אחד מחמשת ה–EmployeeID מארה"ב, מקבצת הזמנות אלו לפי CustomerID, ומחזירה CustomerID שבקבוצת ההזמנות שלהם חמישה ערכי EmployeeID מובחנים (עבור כל חמשת העובדים).

כדי להפוך את הפתרון לדינמי יותר ולהחזיק רשימות של EmployeeID שאינם ידועים מראש, וכן רשימות גדולות אפילו כאשר הם ידועים, ניתן להשתמש בתת–שאילתות במקום ברשימה מפורשת:

```
SELECT CustomerID
FROM dbo.Orders
WHERE EmployeeID IN
  (SELECT EmployeeID FROM dbo.Employees WHERE Country = N'USA')
GROUP BY CustomerID
HAVING COUNT(DISTINCT EmployeeID) =
  (SELECT COUNT(*) FROM dbo.Employees WHERE Country = N'USA');
```

בעיה נוספת המערבת תת–שאילתות עצמאיות היא החזרת כל ההזמנות שנפתחו ביום ההזמנה האחרון בחודש בפועל. שים לב שיום ההזמנה האחרון בחודש בפועל עשוי להיות שונה מהיום האחרון בחודש – למשל, אם חברה אינה מקבלת הזמנות בסופי שבוע. כך שיש לאחזר את יום ההזמנה האחרון בחודש בפועל מתוך הנתונים.

הנה הפתרון בשאילתה שמפיקה את הפלט (מקוצר) המוצג בטבלה 2–4:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate IN
  (SELECT MAX(OrderDate)
   FROM dbo.Orders
   GROUP BY CONVERT(CHAR(6), OrderDate, 112));
```

טבלה 2–4: הזמנות שנפתחו ביום ההזמנה האחרון בחודש בפועל

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*
10461	LILAS	1	1997-02-28
10616	GREAL	1	1997-07-31
10916	RANCH	1	1998-02-27
11077	RATTC	1	1998-05-06
10368	ERNSH	2	1996-11-29
10553	WARTH	2	1997-05-30
10583	WARTH	2	1997-06-30
10686	PICCO	2	1997-09-30
10915	TORTU	2	1998-02-27
10989	QUEDE	2	1998-03-31
…			
11075	RICSU	8	1998-05-06
10687	HUNGO	9	1997-09-30

תת–השאילתה העצמאית מחזירה רשימה של ערכים עם יום ההזמנה האחרון בפועל של כל חודש, כפי שניתן לראות בטבלה 3–4.

טבלה 3–4: יום ההזמנה האחרון בפועל של כל חודש

MAX(OrderDate)
1997-04-30
1996-09-30
1996-10-31
1998-02-27

MAX(OrderDate)
1997-06-30
1997-08-29
1997-03-31
1997-01-31
1997-09-30
1996-07-31
1997-02-28
1998-01-30
1997-11-28
1998-04-30
1997-05-30
1997-12-31
1998-05-06
1997-10-31
1996-11-29
1996-12-31
1996-08-30
1997-07-31
1998-03-31

תת–השאילתה משיגה תוצאה זו על ידי קיבוץ ההזמנות לפי חודש והחזרת MAX(OrderDate) עבור כל קבוצה. חילצתי את החלק של שנה–וחודש מתוך OrderDate על ידי המרתו ל–CHAR(6) תוך שימוש בפרמטר פורמט 112 (תבנית ISO 'YYYYMMDD'). מכיוון שמחרוזת היעד קצרה יותר מהמחרוזת של סגנון זה (6 תווים במקום 8), שני התווים הימניים–קיצוניים נחתכים. באפשרותך להשיג את אותה תוצאה על ידי שימוש ב– YEAR(OrderDate), MONTH(OrderDate) בפסוקית GROUP BY.

השאילתה החיצונית מחזירה את כל ההזמנות עם OrderDate המופיע ברשימה המוחזרת מתת–השאילתה.

תת–שאילתות תלויות

תת–שאילתות תלויות (Correlated Subqueries) הן תת–שאילתות המתייחסות לטורים מהשאילתה החיצונית. לוגית, תת–השאילתה מוערכת פעם אחת עבור כל שורה מהשאילתה

החיצונית. שוב, פיסית, זהו תהליך דינמי הרבה יותר אשר ישתנה ממקרה למקרה. יש יותר מדרך פיסית אחת לעבד תת–שאילתה תלויה.

שובר–שוויון

אתחיל לדון בתת–שאילתות תלויות דרך בעיה המציגה מושג חשוב מאוד בשאילתות SQL – **שובר–שוויון** (Tiebreaker). מושג זה ישוב ויעלה במהלך הספר. שובר–שוויון הוא מאפיין, או רשימת מאפיינים המאפשרים לך לדרג אלמנטים בצורה ייחודית. למשל, נניח שאתה צריך את ההזמנה האחרונה ביותר עבור כל עובד. אתה אמור להחזיר הזמנה אחת בלבד עבור כל עובד, אך המאפיינים EmployeeID ו–OrderDate לא בהכרח מזהים הזמנה יחידה. עליך להוסיף שובר–שוויון כדי לזהות הזמנה אחרונה יחידה עבור כל עובד. למשל, מתוך מספר הזמנות של עובד שלהן OrderDate מקסימלי תוכל להחליט להחזיר את זו שלה OrderID מקסימלי. במקרה זה, MAX(OrderID) הוא שובר–השוויון שלך. או שתוכל להחליט להחזיר את השורה עם RequiredDate מקסימלי, ואם עדיין תקבל יותר משורה אחת, תחזיר את זו שלה OrderID מקסימלי. במקרה זה, שובר–השוויון שלך הוא MAX(OrderID), MAX(RequiredDate). שובר–שוויון אינו מוגבל בהכרח למאפיין בודד. בטרם נמשיך לפתרונות, הרץ את הקוד הבא כדי ליצור אינדקסים התומכים בעיבוד הפיסי של השאילתות בהמשך:

```
CREATE UNIQUE INDEX idx_eid_od_oid
  ON dbo.Orders(EmployeeID, OrderDate, OrderID);
CREATE UNIQUE INDEX idx_eid_od_rd_oid
  ON dbo.Orders(EmployeeID, OrderDate, RequiredDate, OrderID);
```

את הקווים המנחים בנושא אינדקסים אסביר לאחר שאציג את שאילתות הפתרון.

הבה נתחיל עם הבקשה הבסיסית, להחזיר את ההזמנות עם OrderDate מקסימלי לכל עובד. כאן אפשר שתקבל שורות מרובות לכל עובד מכיוון שייתכן כי לעובד מספר הזמנות עם אותו תאריך הזמנה.

ייתכן כי תתפתה להשתמש בפתרון זה, המכיל שאילתה עצמאית דומה לזו בה השתמשנו להחזרת שורות ביום ההזמנה האחרון בחודש בפועל:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders
WHERE OrderDate IN
  (SELECT MAX(OrderDate) FROM dbo.Orders
   GROUP BY EmployeeID);
```

מכל מקום, פתרון זה הוא שגוי. סט התוצאה יכיל את ההזמנות הנכונות (אלו שלהם OrderDate מקסימלי עבור כל עובד). אך הוא יכיל גם כל הזמנה של עובד א' שה– OrderDate שלה הוא במקרה התאריך המקסימלי של עובד ב', אפילו שהוא אינו

התאריך המקסימלי של עובד א'. דבר זה לא היווה בעיה במקרה הקודם, שכן תאריך הזמנה בחודש א' אינו יכול להיות שווה לתאריך ההזמנה המקסימלי של חודש אחר ב'.

במקרה שלנו, תת–השאילתה חייבת להיות תלויה בשאילתה החיצונית, ולקשר את ה–EmployeeID הפנימי לזה שבשורה החיצונית:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderDate =
   (SELECT MAX(OrderDate)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID);
```

שאילתה זו מייצרת את התוצאה הנכונה, המוצגת בטבלה 4–4.

טבלה 4–4: הזמנות בעלות OrderDate מקסימלי לכל עובד

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*	*RequiredDate*
11077	RATTC	1	1998-05-06	1998-06-03
11070	LEHMS	2	1998-05-05	1998-06-02
11073	PERIC	2	1998-05-05	1998-06-02
11063	HUNGO	3	1998-04-30	1998-05-28
11076	BONAP	4	1998-05-06	1998-06-03
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11058	BLAUS	9	1998-04-29	1998-05-27

הפלט מכיל דוגמה אחת של הזמנות מרובות לעובד, במקרה של עובד 2. אם ברצונך להחזיר שורה אחת בלבד לכל עובד, עליך להציג שובר–שוויון. למשל, מתוך השורות המרובות בעלות OrderDate מקסימלי, החזר את זו בעלת OrderID מקסימלי.

דבר זה ניתן להשגה על ידי הוספת תת–שאילתה נוספת השומרת את ההזמנה רק כאשר OrderID שווה למקסימום מבין ההזמנות בעלות EmployeeID ו– OrderDate זהים לאלו שבשורה החיצונית:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderDate =
   (SELECT MAX(OrderDate)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID)
   AND OrderID =
   (SELECT MAX(OrderID)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID
      AND O2.OrderDate = O1.OrderDate);
```

שים לב לפלט המוצג בטבלה 5–4; מתוך שתי ההזמנות של עובד 2 בפלט של השאילתה הקודמת, נשארת רק זו שלה OrderID מקסימלי.

טבלה 5–4: הזמנה אחרונה של כל עובד; שובר–שוויון: Max(OrderID)

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*	*RequiredDate*
11077	RATTC	1	1998-05-06	1998-06-03
11073	PERIC	2	1998-05-05	1998-06-02
11063	HUNGO	3	1998-04-30	1998-05-28
11076	BONAP	4	1998-05-06	1998-06-03
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11058	BLAUS	9	1998-04-29	1998-05-27

במקום להשתמש בשתי תת–שאילתות נפרדות עבור טור המיון (OrderDate) ועבור שובר–השוויון (OrderID), ניתן להשתמש בתת–שאילתות מקוננות:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderID =
   (SELECT MAX(OrderID)
```

```
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID
      AND O2.OrderDate =
        (SELECT MAX(OrderDate)
         FROM dbo.Orders AS O3
         WHERE O3.EmployeeID = O1.EmployeeID));
```

בהשוואת הביצועים של השתיים מצאתי שהן דומות, עם יתרון קל לאחרונה. אני מוצא את גישת הקינון מעט מורכבת יותר, כך שכל עוד אין יתרון משמעותי של ביצועים, אעדיף לדבוק בגישה הפשוטה יותר. פשטות קלה יותר להבנה ולתחזוקה, ולכן פחות מועדת לטעויות.

נחזור לגישה הפשוטה; לכל מאפיין של שובר–שוויון עליך להוסיף תת–שאילתה. כל תת–שאילתה כזו צריכה להיות קשורה לטור הקיבוץ, לטור המיון ולכל מאפייני שובר–השוויון הקודמים. כך שכדי להשתמש ב– MAX(OrderID), MAX(RequiredDate), כשובר–שוויון, עליך לכתוב את השאילתה הבאה:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderDate =
   (SELECT MAX(OrderDate)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID)
   AND RequiredDate =
   (SELECT MAX(RequiredDate)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID
      AND O2.OrderDate = O1.OrderDate)
   AND OrderID =
   (SELECT MAX(OrderID)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID
      AND O2.OrderDate = O1.OrderDate
      AND O2.RequiredDate = O1.RequiredDate);
```

הקו המנחה בנושא אינדקסים עבור שאילתות שובר–השוויון לעיל הוא ליצור אינדקס על (*group_cols*, *sort_cols*, *tiebreaker_cols*). למשל, כאשר שובר–השוויון הוא MAX(OrderID) תרצה אינדקס על (*EmployeeID*, *OrderDate*, *OrderID*).

כאשר שובר–השוויון הוא MAX(OrderID), MAX(RequiredDate), תרצה אינדקס על (*EmployeeID*, *OrderDate*, *RequiredDate*, *OrderID*). אינדקס כזה יאפשר לאחזר את ערך המיון או ערך שובר–השוויון עבור עובד על ידי פעולת חיפוש בתוך האינדקס.

כשתסיים לבחון את הפתרונות לשובר–שוויון, הרץ את הקוד הבא כדי להסיר את האינדקסים שנוצרו עבור דוגמאות אלו:

```
DROP INDEX dbo.Orders.idx_eid_od_oid;
DROP INDEX dbo.Orders.idx_eid_od_rd_oid;
```

הצגתי כאן גישה אחת בלבד לפתרון בעיות שובר–שוויון בעזרת שימוש בתת–שאילתות תלויות תואמות ANSI. גישה זו אינה היעילה או הפשוטה ביותר. תוכל למצוא פתרונות אחרים לבעיות שובר–שוויון בפרק 6 בסעיף "שוברי–שוויון" ובפרק 7 בסעיף "TOP n לכל קבוצה".

EXISTS

EXISTS הוא אופרטור רב עוצמה המאפשר לך לבדוק בצורה יעילה האם שאילתה נתונה תחזיר שורות או לא. הקלט ל–EXISTS הוא תת–שאילתה, לרוב תלויה אך לא בהכרח, והביטוי מחזיר TRUE או FALSE, בהתאם לכך שהשאילתה מחזירה שורה אחת לפחות או שאינה מחזירה שורות. שלא כמו אופרטורים אחרים וביטויים לוגיים, EXISTS אינו יכול להחזיר UNKNOWN. תת–השאילתה של הקלט מחזירה שורות או שאינה מחזירה שורות. אם המסנן של תת–השאילתה מחזיר UNKNOWN עבור שורה מסוימת, השורה אינה מוחזרת. זכור שבמסנן, UNKNOWN מטופל כ–FALSE. במילים אחרות, כאשר לתת–השאילתה של הקלט יש מסנן, EXISTS יחזיר TRUE אך ורק אם המסנן הוא TRUE לפחות עבור שורה אחת. הסיבה שאני מדגיש נקודה זו תתברר מייד.

ראשית, הבא נתבונן בדוגמה שתציג את השימוש ב–EXISTS. השאילתה הבאה מחזירה את כל הלקוחות מספרד שביצעו הזמנות, ומייצרת את הפלט המופיע בטבלה 6–4:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
   AND EXISTS
      (SELECT * FROM Orders AS O
       WHERE O.CustomerID = C.CustomerID);
```

טבלה 4–6: לקוחות מספרד שביצעו הזמנות

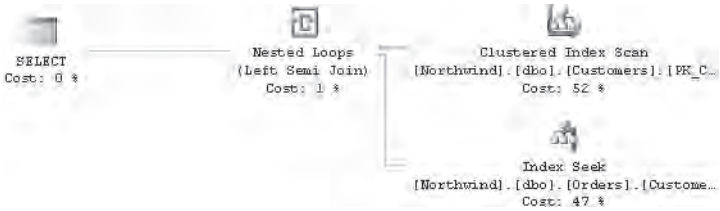
CustomerID	*CompanyName*
BOLID	Bólido Comidas preparadas
GALED	Galería del gastrónomo
GODOS	Godos Cocina Típica
ROMEY	Romero y tomillo

השאילתה החיצונית מחזירה לקוחות מספרד שעבורם האופרטור EXISTS מוצא לפחות שורת הזמנה אחת בטבלת Orders עם CustomerID זהה לזה שבשורת הלקוח החיצונית.

שים לב: השימוש ב–* כאן בטוח לחלוטין, למרות שבאופן עקרוני זהו אינו נוהג מומלץ. ה–optimizer מתעלם מרשימת ה–SELECT המצוינת בתת–השאילתה מכיוון של–EXISTS אכפת רק מעצם קיומן של שורות ולא ממאפיינים מסוימים.

בחן את תוכנית העבודה המיוצרת עבור שאילתה זו, כפי שמוצגת בתרשים 4–1.

תרשים 4–1: תוכנית עבודה עבור שאילתת EXISTS



התוכנית סורקת את טבלת הלקוחות ומסננת לקוחות מספרד. לכל לקוח מתאים, התוכנית מבצעת חיפוש בתוך האינדקס על Orders.CustomerID כדי לבדוק האם טבלת Orders מכילה הזמנה עם ה–CustomerID של לקוח זה. האינדקס על הטור המסונן בתת–השאילתה (Orders.CustomerID במקרה שלנו) מאוד מועיל כאן, מכיוון שהוא מספק גישה ישירה לשורות של טבלת Orders עם ערך CustomerID נתון.

EXISTS לעומת IN

מפתחים תוהים תכופות האם שאילתה עם האופרטור EXISTS יעילה יותר משאילתה מקבילה לוגית עם האופרטור IN. למשל, השאילתה האחרונה יכולה להיכתב תוך שימוש באופרטור IN עם תת–שאילתה עצמאית כלהלן:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
   AND CustomerID IN(SELECT CustomerID FROM dbo.Orders);
```

בגרסאות קודמות ל– SQL Server 2000 זיהיתי הבדלים בין תוכניות המופקות עבור EXISTS ועבור IN, ולרוב ל–EXISTS היו ביצועים טובים יותר בגלל התמיכה ביכולת המעקף (short-circuiting) הפנימי שלו. מכל מקום, ב– SQL Server 2000 ו–2005 ה–optimizer לרוב מייצר תוכניות זהות עבור שתי השאילתות **כאשר הן אכן שוות לוגית**, מה שנכון במקרה זה. התוכנית המיוצרת עבור השאילתה האחרונה המשתמשת ב–IN זהה לזו המופיע בתרשים 1–4, שנוצרה עבור השאילתה המשתמשת ב–EXISTS.

אם אתה מקפיד לחשוב על ההשלכות של הגיון שלושת–הערכים, תבין שישנו הבדל בין IN ל–EXISTS. שלא כמו EXISTS, IN יכול למעשה לייצר תוצאה לוגית UNKNOWN כאשר רשימת הקלט מכילה NULL. למשל, IN(b, c, NULL) הוא UNKNOWN. מכל מקום, מכיוון ש–UNKNOWN מטופל במסנן כ–FALSE, התוצאה של שאילתה עם האופרטור IN זהה לכזו עם האופרטור EXISTS, וה–optimizer מודע לכך, ומכאן התוכניות הזהות.

NOT EXISTS לעומת NOT IN

ההבדל הלוגי בין EXISTS ל–IN מופיע אם משווים NOT EXISTS ל– NOT IN, כאשר רשימת הקלט של NOT IN עשויה להכיל NULL.

למשל, נניח שעליך להחזיר לקוחות מספרד אשר לא ביצעו הזמנות. להלן הפתרון המשתמש באופרטור NOT EXISTS, המפיק את הפלט המוצג בטבלה 7–4:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
   AND NOT EXISTS
      (SELECT * FROM Orders AS O
       WHERE O.CustomerID = C.CustomerID);
```

טבלה 7–4: לקוחות מספרד שלא ביצעו הזמנות

CustomerID	*CompanyName*
FISSA	FISSA Fabrica Inter. Salchichas S.A.

אפילו אם יש CustomerID שהוא NULL בטבלת Orders, הדבר אינו רלוונטי עבורנו. אתה מקבל את כל הלקוחות מספרד שעבורם SQL Server לא יכול למצוא אפילו שורה אחת בטבלת Orders בעלת CustomerID זהה. התוכנית המיוצרת עבור שאילתה זו מוצגת בתרשים 2–4.

תרשים 4–2: תוכנית עבודה עבור שאילתת NOT EXISTS

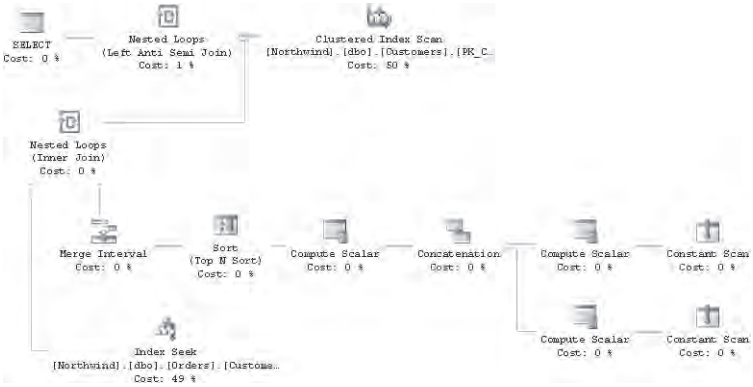
התוכנית סורקת את טבלת Customers ומסננת לקוחות מספרד. לכל לקוח מתאים, התוכנית מבצעת חיפוש בתוך האינדקס על Orders.CustomerID. האופרטור TOP מופיע מכיוון שנחוץ לנו לראות רק האם קיימת ללקוח לפחות הזמנה מתאימה אחת – זוהי יכולת המעקף של EXISTS בפעולה. שימוש זה של TOP יעיל במיוחד כאשר לטור Orders.CustomerID רמת צפיפות גבוהה (כלומר, מספר רב של כפילויות). החיפוש מתבצע פעם אחת בלבד לכל לקוח, וללא קשר למספר ההזמנות שיש ללקוח, שורה אחת בלבד נסרקת ברמת העלה – הרמה התחתונה של האינדקס (leaf level) – בחיפוש אחר התאמה, בניגוד לכל השורות המתאימות.

במקרה זה, הפתרון הבא, המשתמש באופרטור NOT IN אכן מפיק פלט זהה. נדמה שיש לו משמעות זהה, אך מאוחר יותר נראה שלא כך הדבר.

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
   AND CustomerID NOT IN(SELECT CustomerID FROM dbo.Orders);
```

אם תבחן את תוכנית העבודה המוצגת בתרשים 4–3, תמצא שהיא שונה מזו המיוצרת עבור שאילתת NOT EXISTS.

תרשים 4–3: תוכנית עבודה עבור שאילתת NOT IN



יש מספר פעולות נוספות בתחילת תוכנית זו, יחסית לתוכנית הקודמת – צעדים הנדרשים כדי לחפש CustomerID שהם NULL. מדוע תוכנית זו שונה מזו המיוצרת עבור שאילתת NOT EXISTS? ומדוע של– SQL Server יהיה אכפת במיוחד מהקיום של NULLs ב– Orders.CustomerID?

ההבדל בין התוכניות אינו משפיע על התוצאה מכיוון שאין אף שורה בטבלת Orders שלה CustomerID שהוא NULL. יחד עם זאת, מכיוון שהטור CustomerID מתיר NULLs, ה–optimizer חייב להתחשב בעובדה זו. הבה נראה מה קורה אם מוסיפים לטבלת Orders שורה שה– CustomerID שלה הוא NULL:

```
INSERT INTO dbo.Orders DEFAULT VALUES;
```

כעת הרץ שוב את השאילתות NOT EXISTS ו– NOT IN. תמצא שהשאילתה NOT EXISTS עדיין מחזירה את אותו פלט כמקודם, בעוד שהשאילתה NOT IN מחזירה כעת סט ריק. הסיבה לכך היא שהביטוי val IN(val1, val2, …, NULL) לעולם אינו יכול להחזיר FALSE; הוא יכול להחזיר רק TRUE או UNKNOWN. כתוצאה מכך, val **NOT** IN(val1, val2, …, NULL) יכול להחזיר רק NOT TRUE או NOT UNKNOWN, שאף אחד מהם אינו TRUE.

למשל, נניח שרשימת הלקוחות בשאילתה זו היא (a, b, NULL). לקוח a מופיע ברשימה ולפיכך הביטוי a IN(a, b, NULL) מחזיר TRUE. הביטוי a NOT IN(a, b, NULL) מחזיר NOT TRUE, או FALSE, ולקוח a אינו מוחזר על ידי השאילתה. לקוח c, מצד שני, **אינו** מופיע ברשימה (a, b, NULL), אך התוצאה הלוגית של c IN(a,b,NULL) היא UNKNOWN, בגלל ה–NULL. לפיכך הביטוי b NOT IN (a, b, NULL) מחזיר NOT UNKNOWN, אשר שווה ל–UNKNOWN, ולקוח c אינו מוחזר גם על ידי השאילתה, למרות ש–c אינו מופיע ברשימת הלקוחות. בין אם הלקוח מופיע ברשימת הלקוחות או לא, אם זו כוללת NULL, הלקוח לא מוחזר על ידי השאילתה. ברור לך שכאשר קיימת אפשרות ש–NULLs מעורבים (למשל כאשר הטור שעליו מבוצעת השאילתה מאפשר NULLs), NOT EXISTS ו– NOT IN אינם שווים לוגית. דבר זה מסביר את השונות בין התוכניות ואת השוני האפשרי בתוצאות. כדי להפוך את שאילתת NOT IN שווה לוגית לשאילתת EXISTS, הצהר על הטור כ– NOT NULL (אם הדבר אפשרי) או הוסף מסנן לתת–השאילתה כדי להיפטר מ–NULLs:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
   AND CustomerID NOT IN(SELECT CustomerID FROM dbo.Orders
                         WHERE CustomerID IS NOT NULL);
```

השאילתה מייצרת את אותה תוצאה כמו שאילתת NOT EXISTS, וכן את אותה תוכנית עבודה.

כשתסיים להתנסות עם השאילתות, זכור להסיר את השורה שבה CustomerID הוא NULL:

```
DELETE FROM dbo.Orders WHERE CustomerID IS NULL;
```

ערך חסר מינימלי

כדי לתרגל את הידע שלך בנושא האופרטור EXISTS, נסה לפתור את הבעיה הבאה. ראשית צור את טבלה T1 והכנס בה נתונים על ידי הרצת קטע–קוד 1–4.

קטע–קוד 1–4: יצירת טבלה T1 ומילויה בנתונים

```
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  keycol  INT        NOT NULL PRIMARY KEY CHECK(keycol > 0),
  datacol VARCHAR(10)NOT NULL
);
INSERT INTO dbo.T1(keycol, datacol) VALUES(3, 'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(4, 'b');
INSERT INTO dbo.T1(keycol, datacol) VALUES(6, 'c');
INSERT INTO dbo.T1(keycol, datacol) VALUES(7, 'd');
```

שים לב ש–keycol חייב להיות חיובי. משימתך היא לכתוב שאילתה המחזירה את המפתח החסר הנמוך ביותר, בהנחה שערכי המפתח מתחילים ב–1. למשל, בטבלה יש כעת את מפתחות 3, 4, 6, ו–7, לפיכך השאילתה שלך צריכה להחזיר את הערך 1. אם תכניס שתי שורות נוספות, עם המפתחות 1 ו–2, השאילתה שלך צריכה להחזיר 5.

פתרון:

להלן ביטוי CASE מוצע (חלקי) שבו השתמשתי בפתרון שלי:

```
SELECT
   CASE
      WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
      ELSE (...subquery returning minimum missing value...)
   END;
```

אם 1 אינו קיים בטבלה, הביטוי CASE מחזיר 1; אחרת, הוא מחזיר את התוצאה של תת–שאילתה המחזירה את הערך החסר המינימלי.

להלן תת–השאילתה בה השתמשתי כדי להחזיר את הערך החסר המינימלי:

```
SELECT MIN(A.keycol + 1) as missing
FROM dbo.T1 AS A
WHERE NOT EXISTS
   (SELECT * FROM dbo.T1 AS B
    WHERE B.keycol = A.keycol + 1);
```

האופרטור NOT EXISTS מחזיר TRUE רק עבור ערכים בטבלה T1 הנמצאים ממש לפני פער (4 ו–7 במקרה שלנו). ערך נמצא ממש לפני פער אם הערך ועוד 1 אינו קיים באותה טבלה. הכינוי של הטבלה החיצונית T1 הוא A, והכינוי של הטבלה הפנימית T1 הוא B. תוכל להשתמש בביטוי: B.keycol – 1 = A.keycol במסנן של תת–השאילתה; על אף שהשימוש בביטוי שכזה עלול להיות מעט מבלבל כאשר מחפשים ערך ב–B הגדול באחד מהערך ב–A. אם תחשוב על כך, כדי ש–B.keycol יהיה גדול באחד מ–A.keycol, B.keycol פחות אחד חייב להיות שווה ל–A.keycol. אם לוגיקה זו מבלבלת אותך, תוכל להשתמש במקום זאת ב– B.keycol = A.keycol + 1, כפי שאני עשיתי. לאחר שבודדנו את כל הנקודות לפני הפערים, השאילתה החיצונית מחזירה את המינימום ועוד אחד, שהוא הערך החסר הראשון בפער הראשון. השיטה למציאת נקודה לפני פער היא שיטה שתרצה לזכור שכן היא שיטת–מפתח מאוד שימושית.

כעת תוכל לשלב את השאילתה המחזירה את הערך החסר המינימלי בביטוי CASE:

```
SELECT
   CASE
      WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
      ELSE (SELECT MIN(keycol + 1)
            FROM dbo.T1 AS A
            WHERE NOT EXISTS
              (SELECT *
               FROM dbo.T1 AS B
               WHERE B.keycol = A.keycol + 1))
   END;
```

אם תריץ שאילתה זו עם נתוני הדוגמה שנשתלו על ידי קטע–קוד 1–4, התוצאה שתקבל היא 1. אם אז תוסיף שתי שורות, עם מפתחות 1 ו–2 (כפי שנראה בקוד הבא), ותריץ את השאילתה מחדש, התוצאה שתקבל היא 5.

```
INSERT INTO dbo.T1(keycol, datacol) VALUES(1, 'e');
INSERT INTO dbo.T1(keycol, datacol) VALUES(2, 'f');
```

להלן דוגמה כיצד היית יכול להשתמש בביטוי CASE עבור הערך החסר המינימלי במשפט INSERT … SELECT, למשל במקרה בו נדרשת לשוב ולהשתמש במפתחות מחוקים:

```
INSERT INTO dbo.T1(keycol, datacol)
  SELECT
     CASE
        WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
        ELSE (SELECT MIN(keycol + 1)
              FROM dbo.T1 AS A
              WHERE NOT EXISTS
                (SELECT *
                 FROM dbo.T1 AS B
                 WHERE B.keycol = A.keycol + 1))
     END,
     'f';
```

בצע שאילתה על הטבלה T1 לאחר שהרצת את ה–INSERT הזה, ושים לב בפלט המוצג בטבלה 8–4 שפעולת ההכנסה ייצרה את ערך המפתח 5, אשר היה הערך החסר המינימלי:

```
SELECT * FROM dbo.T1;
```

טבלה 8–4: התוכן של T1 לאחר הוספת הנתונים

keycol	*datacol*
1	e
2	f
3	a
4	b
5	**f**
6	c
7	d

שים לב: תהליכים רבים המריצים קוד כזה במקביל עשויים לקבל אותו מפתח. ניתן להתגבר על בעיה זו על ידי הצגת קוד טיפול–בשגיאות שלוכד שגיאת מפתח כפול ואז מנסה שנית. יש שיטות אחרות יעילות יותר לשימוש חוזר של מפתחות מחוקים, אך הן מורכבות יותר ודורשות תחזוקה של טבלה המכילה תחומי ערכים חסרים. כמו כן שים לב ששימוש חוזר במפתחות מחוקים אינו תמיד רעיון טוב, מסיבות שמעבר לבו–זמניות. כאן רק רציתי לתת לך הזדמנות לתרגל את האופרטור EXISTS.

שים לב שביכולתך למזג את שני המקרים, כאשר 1 קיים בטבלה, וכאשר אינו קיים, במקום להשתמש בביטוי CASE. הפתרון דורש תכסיס של מניפולציה לוגית:

```
SELECT COALESCE(MIN(keycol + 1), 1)
FROM dbo.T1 AS A
WHERE NOT EXISTS
      (SELECT * FROM dbo.T1 AS B
         WHERE B.keycol = A.keycol + 1)
      AND EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1);
```

לשאילתה יש את שני הביטויים הלוגיים מהביטוי CASE בפסוקית WHERE. היא מחזירה את הערך החסר המינימלי אם 1 קיים בטבלה (כלומר, כאשר האופרטור EXISTS השני הוא תמיד TRUE). אם 1 אינו קיים בטבלה (כלומר, האופרטור EXISTS השני הוא תמיד FALSE), המסנן מייצר סט ריק והביטוי MIN(keycol) + 1 מניב NULL. הערך של הביטוי COALESCE הוא אז 1.

על אף שפתרון זה משיג את התוצאה בשאילתה אחת, אני אישית מעדיף את הפתרון המקורי. פתרון זה מכיל טריק והוא אינו אינטואיטיבי כמו הקודם; אין להקל ראש בחשיבות של קוד פשוט וקריא.

הפעלת לוגיקה הפוכה על בעיות של חלוקה רלציונית

המוח שלנו רגיל לרוב לחשוב על דרך החיוב. אך חשיבה כזו עלולה להגביל אותך. בתחומים רבים, ותכנות SQL ביניהם, חשיבה על דרך השלילה או לוגיקה הפוכה יכולה לתת לך תובנה חדשה או לשמש אותך ככלי נוסף לפתרון בעיות. הפעלת לוגיקה הפוכה יכולה במקרים מסוימים להוביל לפתרונות פשוטים או יעילים יותר מאשר הפעלת גישה ישירה. זהו כלי נוסף בארגז הכלים שלך.

אאוקלידס, למשל, חיבב מאוד שימוש בלוגיקה הפוכה להוכחות המתמטיות שלו (הוכחה על דרך השלילה). הוא השתמש בלוגיקה הפוכה כדי להוכיח שקיים מספר אינסופי של מספרים ראשוניים. על ידי סתירת הנחה מסוימת, יצירת פרדוקס, אתה מוכיח שההיפך חייב להיות נכון.

לפני שאדגים יישום של לוגיקה הפוכה ב–SQL, הייתי רוצה להעביר את הרעיון דרך חידה עתיקה. שני שומרים עומדים לפני שתי דלתות. דלת אחת מובילה לזהב ואוצרות, והדלת השנייה מובילה למוות פתאומי, אך אתה אינך יודע איזו מהדלתות מובילה לאן. אחד מהשומרים תמיד אומר את האמת והשני תמיד משקר, אך אתה אינך יודע מי השקרן ומי הכן (למרות שהשומרים יודעים). כמובן שברצונך להיכנס בדלת המובילה לזהב ואוצרות ולא למוות פתאומי. לפניך אפשרות אחת לשאול את אחד מהשומרים שאלה; מה תהיה השאלה?

כל שאלה שתשאל בצורה ישירה לא תיתן לך 100 אחוז ביטחון שתבחר את הדלת המובילה לזהב. אך הפעלת לוגיקה הפוכה יכולה לספק לך ביטחון כזה.

שאל אחד מהשומרים, "אם אשאל את השומר השני היכן הדלת שמובילה לזהב, על איזו דלת הוא יצביע?"

אם שאלת את השומר הכן, הוא יצביע על הדלת המובילה למוות פתאומי, שכן הוא יודע שהשני שקרן. אם שאלת את השקרן, גם הוא יצביע על הדלת המובילה למוות פתאומי, שכן הוא יודע שהשומר השני כן ויצביע על הדלת המובילה לזהב. כל שיהיה עליך לעשות הוא להיכנס בדלת עליה לא הצביעו.

לוגיקה הפוכה היא לעיתים כלי שימושי בפתרון בעיות עם SQL. דוגמה לבעיות שתוכל לפתור על ידי שימוש בלוגיקה הפוכה היא בעיות של חלוקה רלציונית. בתחילת הפרק דנתי בבעיה הבאה: מתוך מסד הנתונים Northwind, החזר את כל הלקוחות שההזמנות שלהם טופלו על ידי כל העובדים מארה"ב. הדוגמה שהצגתי לפתרון הבעיה הפעילה חשיבה על דרך החיוב. כדי להפעיל לוגיקה הפוכה, תחילה עליך להצליח לנסח את הבקשה על דרך השלילה. במקום לומר "החזר לקוחות שעבורם כל העובדים מארה"ב טיפלו בהזמנות", תוכל לומר "החזר לקוחות שעבורם אין אף עובד מארה"ב שלא טיפל בהזמנות". זכור שפעמיים לא יוצר כן. אם עבור לקוח א' אינך יכול למצוא אפילו עובד אחד מארה"ב שלא טיפל בשום הזמנה, הרי שכל העובדים מארה"ב בהכרח טיפלו בהזמנות של לקוח א'. ברגע שניסחת את הבקשה על דרך השלילה, התרגום ל–SQL הוא אינטואיטיבי והוא משתמש בתת–שאילתות תלויות:

```
SELECT * FROM dbo.Customers AS C
WHERE NOT EXISTS
   (SELECT * FROM dbo.Employees AS E
    WHERE Country = N'USA'
       AND NOT EXISTS
          (SELECT * FROM dbo.Orders AS O
           WHERE O.CustomerID = C.CustomerID
              AND O.EmployeeID = E.EmployeeID));
```

כשאתה קורא את השאילתה, היא אכן נשמעת כמו הניסוח באנגלית של השאלה:

```
Return customers
for whom you cannot find
   any employee
   from the USA
   for whom you cannot find
      any order
      placed for the subject customer
      and by the subject employee
```

תקבל את אותם 23 לקוחות, כמו אלה המוצגים בטבלה 1–4, המוחזרים על ידי השאילתה המפעילה את הגישה על דרך החיוב. עם זאת שים לב, שהפתרון ההפוך מציג לך את

כל מאפייני הלקוח, בעוד שהפתרון הישיר מציג לך רק קודי לקוח. כדי לקבל מאפיינים אחרים של הלקוח, תצטרך להוסיף join בין סט התוצאה לטבלת הלקוחות.

כאשר משווים את הביצועים של הפתרונות במקרה זה, לפתרון המפעיל גישה על דרך החיוב ביצועים טובים יותר. במקרים אחרים, הגישה על דרך השלילה עשויה להניב ביצועים טובים יותר. כעת בידך כלי נוסף בו תוכל להשתמש לפתרון בעיות.

שאילתות המתנהגות בצורה לא צפויה

ישנה בעיית תכנות ערמומית מאוד הקשורה לתת–שאילתות בה נתקלתי לעיתים ואפילו היה לי חוסר המזל לכלול אותה בקוד מערכות תפעוליות בעצמי. ראשית אתאר את הבאג, ואז אתן המלצות כיצד להימנע ממנו.

נניח שאתה נדרש להחזיר מובילים ממסד הנתונים Northwind שלא הובילו הזמנות ללקוח LAZYK. בבחינת הנתונים, ניתן לראות שמוביל 1 (Speedy Express) הוא היחיד שמתאים להגדרה. השאילתה הבאה אמורה להחזיר את התוצאה הרצויה:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers
WHERE ShipperID NOT IN
   (SELECT ShipperID FROM dbo.Orders
    WHERE CustomerID = N'LAZYK');
```

באופן מפתיע, שאילתה זו מחזירה סט ריק. האם תוכל לומר מדוע? האם תוכל לזהות את הבאג החמקמק בקוד?

ובכן, מסתבר שהטור בטבלת Orders המחזיק את ה–ShipperID נקרא ShipVia ולא ShipperID. בטבלת Orders אין טור שנקרא ShipperID. מתוך הבנה שזה המצב, ודאי תצפה שהשאילתה תכשל בגלל שם טור שגוי. ואכן, אם תריץ רק את החלק שהיה אמור להיות תת–שאילתה עצמאית, הוא ייכשל: שם טור שגוי 'ShipperID'. מכל מקום, בהקשר של השאילתה החיצונית, מסתבר שהשאילתה תקנית! תהליך הבדיקה של שמות הטורים פועל מרמת הקינון הפנימית החוצה. ה– query processor מחפש את הטור ShipperID ראשית בטבלת Orders, אליה פונים ברמה הנוכחית. לאחר שאינו מוצא שם טור כזה, הוא מחפש אותו בטבלת Shippers – הרמה החיצונית – ומוצא אותו. בלי כוונה, תת–השאילתה הפכה להיות תלויה, כאילו הייתה כתובה לפי הדוגמה הבאה:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
   (SELECT S.ShipperID FROM dbo.Orders AS O
    WHERE O.CustomerID = N'LAZYK');
```

לוגית, השאילתה אינה הגיונית כמובן; למרות זאת, השיטה תקינה.

כעת אתה מבין מדוע קיבלת סט ריק. אלא אם כן אין הזמנות ללקוח LAZYK בכלל בטבלת Orders, כמובן שמוביל n תמיד יהיה בסט (SELECT n FROM dbo.Orders WHERE CustomerID = 'LAZYK'), והאופרטור NOT IN תמיד יניב FALSE. שאילתה משובשת–לוגית זו הפכה לשאילתה לבדיקת אי–קיימות המקבילה לקוד בדוגמה הבאה:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers
WHERE NOT EXISTS
   (SELECT * FROM dbo.Orders
    WHERE CustomerID = N'LAZYK');
```

כדי לתקן את הבעיה, עליך להשתמש כמובן בשם הטור הנכון מטבלת Orders המחזיק את ה– ShipVia – ShipperID:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
   (SELECT ShipVia FROM dbo.Orders AS O
    WHERE CustomerID = N'LAZYK');
```

קוד זה יפיק את התוצאה הצפויה המוצגת בטבלה 9–4.

טבלה 9–4: מובילים שלא הובילו הזמנות ללקוח LAZYK

ShipperID	*CompanyName*
1	Speedy Express

מכל מקום, כדי להימנע מבאגים כאלו בעתיד, מומלץ לכלול תמיד את שם הטבלה או את כינויה בכל המאפיינים בתת–שאילתה, אפילו כשתת–השאילתה עצמאית. אם הייתי מוסיף בתת–השאילתה תחילית של שם הטבלה ל–ShipperID (כפי שמופיע בקוד הבא), הייתה מופיעה שגיאת בדיקת שמות טורים והייתי מזהה את הבאג:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
   (SELECT O.ShipperID FROM dbo.Orders AS O
    WHERE O.CustomerID = N'LAZYK');

Msg 207, Level 16, State 1, Line 4
Invalid column name 'ShipperID'.
```

לבסוף, לאחר תיקון הבאג, להלן שאילתת הפתרון כפי שהיא צריכה להיראות:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
   (SELECT O.ShipVia FROM dbo.Orders AS O
    WHERE O.CustomerID = N'LAZYK');
```

אופרטורים לא שגרתיים

בנוסף ל–IN ול–EXISTS, קיימים שלושה אופרטורים נוספים ב–SQL, אך השימוש בהם מתבצע לעיתים רחוקות בלבד: ANY, SOME ו–ALL. ניתן להתייחס אליהם כהכללות של האופרטור IN (ANY ו–SOME הם זהים, ואין שוני לוגי ביניהם).

האופרטור IN מתורגם לסדרה של אופרטורי שוויון מופרדים על ידי אופרטורי OR – למשל, v IN(x, y, z) מתורגם ל– v = x OR v = y OR v = z. ANY (או SOME) מאפשר לך לציין את ההשוואה שאתה רוצה בכל אופרטור, מבלי להגביל אותך לאופרטור השוויון. למשל, v < > ANY(x, y, z) מתורגם ל– v <> x OR v <> y OR v <> z.

ALL דומה, אך הוא מתורגם לסדרה של ביטויים לוגיים המופרדים על ידי אופרטורי AND. למשל, v < > ALL(x, y, z) מתורגם ל– v <> x AND v <> y AND v <> z.

שים לב: IN מאפשר כקלט רשימה מילולית או תת–שאילתה המחזירה טור יחיד. ANY/SOME ו–ALL תומכים רק בתת–שאילתה כקלט. אם עליך להשתמש באופרטורים בלתי שגרתיים אלו עם רשימה מילולית כקלט, עליך להמיר את הרשימה בתת–שאילתה. כך שבמקום v < > ANY(x, y, z), תשתמש ב–

v < > ANY(SELECT x UNION ALL SELECT y UNION ALL SELECT z)

כדי להדגים את השימוש באופרטורים בלתי שגרתיים אלו, הבה נניח שהתבקשת להחזיר, לכל עובד, את ההזמנה בעלת ה–OrderID המינימלי. להלן הדרך בה תוכל לבצע זאת בעזרת האופרטור ANY, אשר יפיק את התוצאה המוצגת בטבלה 10–4:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders AS O1
WHERE NOT OrderID >
   ANY(SELECT OrderID
       FROM dbo.Orders AS O2
       WHERE O2.EmployeeID = O1.EmployeeID);
```

טבלה 10–4: שורה בעלת OrderID מינימלי לכל עובד

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*
10248	VINET	5	1996-07-04
10249	TOMSP	6	1996-07-05
10250	HANAR	4	1996-07-08
10251	VICTE	3	1996-07-08
10255	RICSU	9	1996-07-12
10258	ERNSH	1	1996-07-17
10262	RATTC	8	1996-07-22
10265	BLONP	2	1996-07-25
10289	BSBEV	7	1996-08-26

לשורה יש OrderID מינימלי לעובד, אם ה–OrderID שלה אינו גדול מאף OrderID של אותו עובד.

תוכל לכתוב גם שאילתה המשתמשת ב–ALL להשגת דבר זהה:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders AS O1
WHERE OrderID <=
   ALL(SELECT OrderID
       FROM dbo.Orders AS O2
       WHERE O2.EmployeeID = O1.EmployeeID);
```

לשורה יש OrderID מינימלי לעובד אם ה–OrderID שלה שווה או קטן מכל ה–OrderIDs של אותו עובד.

אף אחד מהפתרונות לעיל לא יפול תחת הקטגוריה של פתרון אינטואיטיבי, וייתכן שהדבר יכול להסביר מדוע אופרטורים אלו אינם נמצאים בשימוש נרחב. נראה כי הדרך הטבעית לכתיבת שאילתת הפתרון היא כלהלן:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders AS O1
WHERE OrderID =
   (SELECT MIN(OrderID)
    FROM dbo.Orders AS O2
    WHERE O2.EmployeeID = O1.EmployeeID);
```

ביטויי טבלה

עד כה כיסיתי תת–שאילתות סקלאריות ומרובות–ערכים. סעיף זה מטפל בתת–שאילתות טבלאיות, הידועות בשם **ביטויי טבלה (Table Expressions)**. בפרק זה, אדון בטבלאות נגזרות (derived tables) ובחידוש של ביטוי טבלה שגור CTE (Common Table Expression).

למידע נוסף על שני הסוגים האחרים של ביטויי טבלה – views ופונקציות מוגדרות משתמש UDF (User Defined Functions) – נא עיין בספר:
"Inside Microsoft SQL Server 200: T-SQL Programming"
(Microsoft Press, 2006).

טבלאות נגזרות

טבלה נגזרת היא ביטוי טבלה – כלומר, טבלת תוצאה וירטואלית הנגזרת מביטוי שאילתה. טבלה נגזרת מופיעה בפסוקית FROM של שאילתה ככל טבלה אחרת. טווח המחיה של טבלה נגזרת הוא אך ורק טווח המחיה של השאילתה החיצונית.

הנוסח הכללי לשימוש בטבלה נגזרת הוא כלהלן:

```
FROM (derived_table_query expression) AS derived_table_alias
```

שים לב: טבלה נגזרת היא וירטואלית לחלוטין. היא אינה ממומשת פיסית, וה–optimizer אף אינו מייצר תוכנית נפרדת עבורה. השאילתות החיצונית והפנימית ממוזגות, ותוכנית אחת מיוצרת. השימוש בטבלאות נגזרות אינו צריך לגרום לך דאגות מיוחדות בנוגע לביצועים. השימוש בטבלאות נגזרות כשלעצמו לא מאט ולא משפר ביצועים. השימוש בהן הוא יותר עניין של פשטות ובהירות של קוד.

טבלה נגזרת חייבת להיות תקינה; לפיכך היא חייבת לעמוד במספר תנאים:

- כל הטורים חייבים להיות בעלי שמות.
- שמות הטורים חייבים להיות ייחודיים.
- ORDER BY אינו מותר (אלא אם כן מוגדר גם TOP).

שים לב: שלא כמו שאילתות סקלאריות או מרובות–ערכים, טבלאות נגזרות אינן יכולות להיות תלויות; הן חייבות להיות עצמאיות. יש יוצא מן הכלל לכלל זה כאשר משתמשים באופרטור APPLY החדש, אותו אסקור בפרק 7.

כינויים לטורי תוצאה

אחד השימושים של טבלאות נגזרות הוא לאפשר שימוש מחדש של כינויי טורים כאשר ביטויים הם כה ארוכים שתעדיף לא לחזור עליהם. לשם הפשטות, אדגים שימוש מחדש בכינויי טורים עם ביטויים קצרים.

כזכור לך מפרק 1, לא ניתן להשתמש בכינויים הנוצרים ברשימת ה–SELECT של השאילתה במרבית מהאלמנטים של השאילתה. הסיבה לך היא שהפסוקית SELECT מעובדת לוגית כמעט אחרונה, מייד לפני הפסוקית ORDER BY. מסיבה זו שאילתת ההדגמה הבאה תיכשל:

```
SELECT
   YEAR(OrderDate) AS OrderYear,
   COUNT(DISTINCT CustomerID) AS NumCusts
FROM dbo.Orders
GROUP BY OrderYear;
```

הפסוקית GROUP BY מעובדת לוגית לפני הפסוקית SELECT, כך שבשלב ה– GROUP BY, הכינוי OrderYear עדיין לא נוצר. על ידי שימוש בטבלה נגזרת המכילה אך ורק את האלמנטים SELECT ו–FROM מהשאילתה המקורית, ניתן ליצור כינויים שיהיו זמינים לשאילתה החיצונית בכל אלמנט.

ישנם שני סגנונות למתן כינויים לטורי התוצאה של טבלה נגזרת. האחד הוא כינוי טור פנימי:

```
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
      FROM dbo.Orders) AS D
GROUP BY OrderYear;
```

והשני הוא כינוי טור חיצוני בהתאם לכינוי של הטבלה הנגזרת:

```
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM (SELECT YEAR(OrderDate), CustomerID
           FROM dbo.Orders) AS D(OrderYear, CustomerID)
GROUP BY OrderYear;
```

לאחרונה אני משתמש לרוב בכינויי טור פנימיים, שכן אני מוצא שזה מעשי הרבה יותר. אינך צריך לציין כינויים לטורי בסיס, ונוח הרבה יותר לזהות שיבושים. כאשר אתה מדגיש ומריץ רק את השאילתה של הטבלה הנגזרת, סט התוצאה שאתה מקבל מכיל את כל שמות טורי התוצאה. כמו כן, ברור איזה כינוי טור שייך לאיזה ביטוי.

לסגנון מתן כינויי טור החיצוני אין את כל היתרונות שהוזכרו. איני יכול לחשוב אפילו על יתרון אחד שיש לו. מכיוון שמתן כינויי טור חיצוני אינו ידע שגור בידי תוכניתני

SQL, ייתכן שתחשוב ששימוש בו מעיד על רמת בקיאות גבוהה ב–SQL. אני יודע שאני חשבתי כך בשלב מסוים. כמובן, שזהו אינו קו מנחה נעלה כמו בהירות של קוד וקלות של תחזוקה ואיתור בעיות.

שימוש בארגומנטים

למרות ששאילתה של טבלה נגזרת אינה יכולה להיות תלויה, היא יכולה להתייחס למשתנים המוגדרים באותה אצווה. למשל, הקוד הבא מחזיר לכל שנה את מספר הלקוחות שטופלו על ידי עובד 3 ומפיק את הפלט המוצג בטבלה 11–4:

```
DECLARE @EmpID AS INT;
SET @EmpID = 3;

SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
      FROM dbo.Orders
      WHERE EmployeeID = @EmpID) AS D
GROUP BY OrderYear;
```

טבלה 11–4: ספירה שנתית של לקוחות שטופלו על ידי עובד 3

OrderYear	*NumCusts*
1996	16
1997	46
1998	30

קינון

טבלאות נגזרות יכולות להיות מקוננות. עיבוד לוגי במקרה של טבלאות נגזרות מקוננות מתחיל ברמה הפנימית ביותר וממשיך כלפי חוץ.

השאילתה הבאה (המפיקה את הפלט המוצג בטבלה 12–4) מחזירה את שנת ההזמנה ואת מספר הלקוחות לשנים בהן למעלה מ–70 לקוחות פעילים:

```
SELECT OrderYear, NumCusts
FROM (SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
      FROM (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
            FROM dbo.Orders) AS D1
      GROUP BY OrderYear) AS D2
WHERE NumCusts > 70;
```

טבלה 4–12: שנת הזמנה ומספר לקוחות לשנים בהן למעלה מ–70 לקוחות פעילים

OrderYear	*NumCusts*
1997	86
1998	81

התייחסויות מרובות

מתוך כל הסוגים של ביטויי הטבלה הקיימים ב–T-SQL, טבלאות נגזרות הן הסוג היחיד אשר לו מגבלה הקשורה להתייחסויות מרובות. אינך יכול להתייחס לאותה טבלה נגזרת מספר פעמים באותה שאילתה. למשל, נניח שברצונך להשוות את מספר הלקוחות הפעילים בכל שנה למספר הלקוחות הפעילים בשנה הקודמת. ברצונך לאחד שני מופעים של טבלה נגזרת המכילה את הסיכומים השנתיים. במקרה כזה, למרבה הצער, עליך לייצר שתי טבלאות נגזרות, כל אחת חוזרת על אותה שאילתה של טבלה נגזרת:

```
SELECT Cur.OrderYear,
   Cur.NumCusts AS CurNumCusts, Prv.NumCusts AS PrvNumCusts,
   Cur.NumCusts - Prv.NumCusts AS Growth
FROM (SELECT YEAR(OrderDate) AS OrderYear,
        COUNT(DISTINCT CustomerID) AS NumCusts
      FROM dbo.Orders
      GROUP BY YEAR(OrderDate)) AS Cur
   LEFT OUTER JOIN
     (SELECT YEAR(OrderDate) AS OrderYear,
        COUNT(DISTINCT CustomerID) AS NumCusts
      FROM dbo.Orders
      GROUP BY YEAR(OrderDate)) AS Prv
    ON Cur.OrderYear = Prv.OrderYear + 1;
```

הפלט של שאילתה זו מוצג בטבלה 4–13.

טבלה 4–13: השוואת מספר לקוחות בשנה נוכחית לשנה הקודמת

OrderYear	*CurNumCusts*	*PrvNumCusts*	*Growth*
1996	67	NULL	NULL
1997	86	67	19
1998	81	86	-5

ביטויי טבלה שגורים (CTE)

ביטוי טבלה שגור (CTE) הוא סוג חדש של ביטוי טבלה שנכנס לשימוש ב– SQL Server 2005. באשר לתקן, CTE הוצג במפרט של ANSI SQL:1999. מבחינות רבות, תמצא ש–CTE מאוד דומה לטבלה נגזרת. אך ל–CTE מספר יתרונות חשובים, אותם אתאר בסעיף זה.

להזכירך, טבלה נגזרת מופיעה בשלמותה בפסוקית FROM של שאילתה חיצונית. CTE, לעומת זאת, מוגדר לראשונה תוך שימוש במשפט WITH, ושאילתה חיצונית המתייחסת לשם ה–CTE ממשיכה את ההגדרה של ה–CTE:

```
WITH cte_name
AS
(
  cte_query
)
outer_query_referring to_cte_name;
```

שים לב: מכיוון שלמילת המפתח WITH שימושים נוספים ב–T-SQL, כדי למנוע דו–משמעיות, המשפט המקדים את הפסוקית WITH של ה–CTE חייב להסתיים בתו נקודה–פסיק. השימוש בתו נקודה–פסיק לסיום משפטים נתמך על ידי ANSI. זהו נוהג רצוי, וכדאי להתחיל להשתמש בו אפילו היכן ש–T-SQL כרגע לא מחייב זאת.

טווח הקיום של ה–CTE הוא טווח המחייה של השאילתה החיצונית. הוא אינו נראה על ידי משפטים אחרים באותה אצווה.

אותם כללים שהזכרתי לצורך החוקיות של ביטוי שאילתה של טבלה נגזרת חלים גם על CTE. כלומר, השאילתה חייבת לייצר טבלה חוקית, כך שלכל הטורים חייב להיות שם, כל שמות הטורים חייבים להיות ייחודיים, ו– ORDER BY אינו מותר (אלא אם כן מוגדר גם TOP).

כעת, אעבור על אספקטים של CTE, תוך כדי הצגת התחביר שלהם ויכולותיהם, ואשווה אותם לטבלאות נגזרות.

כינויים של טורי תוצאה

כפי שניתן עם טבלאות נגזרות, ביכולתך לתת כינויים לטורי תוצאה, פנימית בשאילתה של ה–CTE או חיצונית בסוגריים לאחר שם ה–CTE.

הקוד הבא מדגים את השיטה הראשונה:

```
WITH C AS
(
  SELECT YEAR(OrderDate) AS OrderYear, CustomerID
  FROM dbo.Orders
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM C
GROUP BY OrderYear;
```

קטע הקוד הבא מדגים כיצד לתת כינויים חיצונית בסוגריים לאחר שם ה–CTE:

```
WITH C(OrderYear, CustomerID) AS
(
  SELECT YEAR(OrderDate), CustomerID
  FROM dbo.Orders
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM C
GROUP BY OrderYear;
```

שימוש בארגומנטים

דמיון נוסף בין CTE לטבלאות נגזרות, הוא ש–CTE יכול להתייחס למשתנים שהוצגו באותה אצווה:

```
DECLARE @EmpID AS INT;
SET @EmpID = 3;

WITH C AS
(
  SELECT YEAR(OrderDate) AS OrderYear, CustomerID
  FROM dbo.Orders
  WHERE EmployeeID = @EmpID
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM C
GROUP BY OrderYear;
```

CTEs מרובים

שלא כמו טבלאות נגזרות, CTEs אינם יכולים להיות מקוננים ישירות. כלומר, לא ניתן להגדיר CTE בתוך CTE אחר. אף על פי כן, ניתן להגדיר CTEs מרובים על ידי שימוש באותו משפט WITH, כאשר כל אחד מהם יכול להתייחס ל–CTEs הקודמים. לשאילתה החיצונית יש גישה לכל אחד מה–CTEs. על ידי שימוש ביכולת זו, ניתן להשיג את אותה תוצאה המושגת על ידי קינון טבלאות נגזרות. למשל, משפט WITH הבא מגדיר שני CTEs:

```
WITH C1 AS
(
  SELECT YEAR(OrderDate) AS OrderYear, CustomerID
  FROM dbo.Orders
),
C2 AS
(
  SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
  FROM C1
  GROUP BY OrderYear
)
SELECT OrderYear, NumCusts
FROM C2
WHERE NumCusts > 70;
```

C1 מחזיר שנת הזמנה וקוד לקוח לכל הזמנה, ומייצר את הכינוי OrderYear עבור שנת ההזמנה. C2 מקבץ את השורות שהוחזרו מ–C1 לפי OrderYear וצובר את כמות ה–CustomerID הייחודיים (מספר לקוחות פעילים). לבסוף, השאילתה החיצונית מחזירה רק שנות הזמנה בהן למעלה מ–70 לקוחות פעילים.

התייחסויות מרובות

אחד מהיתרונות שיש ל–CTE על פני טבלאות נגזרות הוא שניתן להתייחס לאותו CTE מספר פעמים בשאילתה החיצונית. אין צורך לחזור על אותה הגדרת CTE כפי שנעשה בטבלאות נגזרות. למשל, הקוד הבא מדגים פתרון CTE לבקשה להשוות את מספר הלקוחות הפעילים בכל שנה לזה של השנה שעברה:

```
WITH YearlyCount AS
(
  SELECT YEAR(OrderDate) AS OrderYear,
     COUNT(DISTINCT CustomerID) AS NumCusts
  FROM dbo.Orders
```

```
  GROUP BY YEAR(OrderDate)
)
SELECT Cur.OrderYear,
  Cur.NumCusts AS CurNumCusts, Prv.NumCusts AS PrvNumCusts,
  Cur.NumCusts - Prv.NumCusts AS Growth
FROM YearlyCount AS Cur
  LEFT OUTER JOIN YearlyCount AS Prv
     ON Cur.OrderYear = Prv.OrderYear + 1;
```

ניתן לראות שהשאילתה החיצונית מתייחסת ל– YearlyCount CTE פעמיים – פעם כמייצג את השנה הנוכחית (Cur), ופעם כמייצג את השנה הקודמת (Prv).

שינוי נתונים

ניתן לשנות נתונים דרך CTE. להדגמת יכולת זו, ראשית הרץ את הקוד בקטע–קוד 2–4 כדי ליצור את הטבלה CustomersDups ולהכניס בה נתונים לדוגמה.

קטע–קוד 2–4: יצירת טבלה CustomersDups ומילויה בנתונים

```
IF OBJECT_ID('dbo.CustomersDups') IS NOT NULL
  DROP TABLE dbo.CustomersDups;
GO

WITH CrossCustomers AS
(
  SELECT 1 AS c, C1.*
  FROM dbo.Customers AS C1, dbo.Customers AS C2
)
SELECT ROW_NUMBER() OVER(ORDER BY c) AS KeyCol,
  CustomerID, CompanyName, ContactName, ContactTitle, Address,
  City, Region, PostalCode, Country, Phone, Fax
INTO dbo.CustomersDups
FROM CrossCustomers;

CREATE UNIQUE INDEX idx_CustomerID_KeyCol
  ON dbo.CustomersDups(CustomerID, KeyCol);
```

שים לב שהשתמשתי כאן בפונקציה חדשה הנקראת ROW_NUMBER כדי ליצור סדרת מספרים שלמים שישמשו כמפתחות ייחודיים. ב– SQL Server 2000, ניתן להשיג זאת על ידי שימוש בפונקציה IDENTITY במשפט SELECT INTO. בהמשך הפרק אדון בהרחבה על פונקציה זו.

למעשה, הקוד בקטע–קוד 2–4 יוצר טבלת לקוחות עם מופעים כפולים רבים של כל לקוח. הקוד הבא מדגים כיצד ניתן להסיר כפילויות של לקוחות על ידי שימוש ב–CTE.

```
WITH JustDups AS
(
  SELECT * FROM dbo.CustomersDups AS C1
  WHERE KeyCol <
    (SELECT MAX(KeyCol) FROM dbo.CustomersDups AS C2
     WHERE C2.CustomerID = C1.CustomerID)
)
DELETE FROM JustDups;
```

ה– JustDups CTE מכיל את כל השורות המשוכפלות של כל לקוח, אך לא את השורה בה KeyCol מקסימלי עבור הלקוח. שים לב שהקוד בקטע–קוד 2–4 יוצר אינדקס על (CustomerID, KeyCol) כדי לתמוך במסנן. כל שהשאילתה החיצונית עושה הוא למחוק את כל השורות מ–JustDups. לאחר שקוד זה רץ, הטבלה CustomersDups מכילה רק שורות ייחודיות. בשלב זה, ניתן ליצור מפתח ראשי או מגבלה ייחודית על הטור CustomerID כדי להימנע מכפילויות בעתיד.

אובייקטים מכילים

ניתן להשתמש ב–CTE באובייקטים מכילים כגון views ו–UDF פנימיים. יכולת זו מספקת עיטוף (encapsulation), מושג חשוב בתכנות מודולארי. בנוסף לכך, הזכרתי קודם ש–CTEs אינם יכולים להיות מקוננים ישירות. יחד עם זאת, ניתן לקנן CTE בצורה עקיפה על ידי עיטוף CTE באובייקט מכיל וביצוע שאילתה על האובייקט המכיל מ–CTE חיצוני.

שימוש ב–CTE ב–views או ב–UDF פנימיים הוא מאוד טריוויאלי. הדוגמה הבאה יוצרת view המחזיר ספירה שנתית של לקוחות:

```
CREATE VIEW dbo.VYearCnt
AS
WITH YearCnt AS
(
  SELECT YEAR(OrderDate) AS OrderYear,
    COUNT(DISTINCT CustomerID) AS NumCusts
  FROM dbo.Orders
  GROUP BY YEAR(OrderDate)
)
SELECT * FROM YearCnt;
GO
```

שאילתה המבוצעת על ה–view, כפי שמוצג בקוד הבא, מחזירה את הפלט המוצג בטבלה 4–14:

```
SELECT * FROM dbo.VYearCnt;
```

טבלה 4–14: ספירה שנתית של לקוחות

OrderYear	*NumCusts*
1996	67
1997	86
1998	81

אם ברצונך להעביר ארגומנט קלט לאובייקט המכיל למשל, החזרת ספירה שנתית של לקוחות לעובד נתון – תוכל ליצור UDF פנימי כלהלן:

```
CREATE FUNCTION dbo.fn_EmpYearCnt(@EmpID AS INT) RETURNS TABLE
AS
RETURN
  WITH EmpYearCnt AS
  (
    SELECT YEAR(OrderDate) AS OrderYear,
      COUNT(DISTINCT CustomerID) AS NumCusts
    FROM dbo.Orders
    WHERE EmployeeID = @EmpID
    GROUP BY YEAR(OrderDate)
  )
  SELECT * FROM EmpYearCnt;
GO
```

שאילתה המבוצעת על ה–UDF ואספקת עובד 3 כקלט מחזירה את הפלט המוצג בטבלה 4–15:

```
SELECT * FROM dbo.fn_EmpYearCnt(3);
```

טבלה 4–15: ספירה שנתית של לקוחות

OrderYear	*NumCusts*
1996	16
1997	46
1998	30

CTEs רקורסיביים

CTEs רקורסיביים מייצגים את אחד מחידושי T-SQL המשמעותיים ביותר ב– SQL Server 2005. סוף כל סוף, SQL Server תומך ביכולות שאילתה רקורסיבית עם שאילתות מבוססות–סטים טהורות. סוג המשימות והפעילויות להן תורמות שאילתות רקורסיביות כוללות מניפולציה של גרפים, עצים, היררכיות, ורבות אחרות. כעת אציג לך רק CTEs רקורסיביים. מידע נוסף ויישומים מפורטים ניתן למצוא בכיסוי נרחב בפרק 9.

אתאר CTE רקורסיבי על ידי שימוש בדוגמה. ניתן לך EmployeeID כקלט (למשל, עובד 5) מטבלת העובדים במסד הנתונים Northwind. עליך להחזיר את עובד הקלט ואת העובדים הכפופים לו בכל הרמות, בהתבסס על היחסים ההיררכיים המתוחזקים על ידי המאפיינים EmployeeID ו–ReportsTo. המאפיינים אותם עליך להחזיר לכל עובד כוללים: EmployeeID, ReportsTo, FirstName ו–LastName.

בטרם אדגים ואסביר את הקוד של CTE רקורסיבי, אצור אינדקס–כיסוי (Covering Index) שיהיה אופטימלי למשימה:

```
CREATE UNIQUE INDEX idx_mgr_emp_ifname_ilname
  ON dbo.Employees(ReportsTo, EmployeeID)
  INCLUDE(FirstName, LastName);
```

אינדקס זה יאפשר להביא כפיפים ישירים של כל מנהל על ידי שימוש בחיפוש יחיד ובנוסף סריקה חלקית. שים לב לטורים הנכללים (FirstName ו–LastName) שנוספו למטרות כיסוי.

להלן הקוד של ה–CTE הרקורסיבי שיחזיר את התוצאה המבוקשת, המוצגת בטבלה 16–4:

```
WITH EmpsCTE AS
(
  SELECT EmployeeID, ReportsTo, FirstName, LastName
  FROM dbo.Employees
  WHERE EmployeeID = 5

  UNION ALL

  SELECT EMP.EmployeeID, EMP.ReportsTo, EMP.FirstName, EMP.LastName
  FROM EmpsCTE AS MGR
    JOIN dbo.Employees AS EMP
      ON EMP.ReportsTo = MGR.EmployeeID
)
SELECT * FROM EmpsCTE;
```

טבלה 16–4: כפיפים של עובד 5

EmployeeID	*ReportsTo*	*FirstName*	*LastName*
5	2	Steven	Buchanan
6	5	Michael	Suyama
7	5	Robert	King
9	5	Anne	Dodsworth

CTE רקורסיבי מכיל לפחות שתי שאילתות (ידועות גם בכינוי members). השאילתה הראשונה המופיעה בגוף ה–CTE ידועה בשם **איבר העוגן** (Anchor Member). איבר העוגן הוא בסך הכל שאילתה המחזירה טבלה חוקית והוא משמש כבסיס או כעוגן (anchor) לרקורסיה. במקרה שלנו, איבר העוגן מחזיר פשוט את השורה עבור עובד הקלט (עובד 5). השאילתה השנייה המופיעה בגוף ה–CTE ידועה **כאיבר הרקורסיבי** (Recursive Member). מה שהופך את השאילתה לאיבר רקורסיבי היא פנייה רקורסיבית לשם ה– EmpsCTE – CTE. שים לב שפנייה זו אינה דומה לפנייה לשם ה–CTE בשאילתה החיצונית. הפנייה בשאילתה החיצונית מקבלת את טבלת התוצאה הסופית המוחזרת על ידי ה–CTE, ואינה מערבת רקורסיה כלל. לעומת זאת, הפנייה הפנימית מתבצעת בטרם מושלמת טבלת התוצאה של ה–CTE, והיא אלמנט המפתח שמפעיל את הרקורסיה. בצורה כללית ניתן לומר שפנייה פנימית זו לשם ה–CTE מייצגת את "סט התוצאה הקודם". בהפעלה הראשונה של האיבר הרקורסיבי, הפנייה לשם ה–CTE מייצגת את סט התוצאה המוחזר מאיבר העוגן. במקרה שלנו, האיבר הרקורסיבי מחזיר כפיפים של העובדים המוחזרים בסט התוצאה הקודם – במילים אחרות, הרמה הבאה של עובדים.

לא קיימת בדיקת עצירה מפורשת של הרקורסיה; אלא שהרקורסיה עוצרת ברגע שהאיבר הרקורסיבי מחזיר סט ריק. מכיוון שההפעלה הראשונה של האיבר הרקורסיבי הפיקה סט לא–ריק (עובדים 6, 7 ו–9), הוא מופעל שוב. בפעם השנייה שהאיבר הרקורסיבי מופעל, הפנייה לשם ה–CTE מייצגת את סט התוצאה המוחזר על ידי ההפעלה הקודמת של האיבר הרקורסיבי (עובד 6, 7 ו–9). מכיוון שלעובדים אלו אין כפיפים, ההפעלה השנייה של האיבר הרקורסיבי מפיקה סט ריק, והרקורסיה נעצרת.

הפנייה לשם ה–CTE בשאילתה החיצונית מייצגת את האיחוד (שרשור) של כל הסטים של התוצאות מהפעלת איבר העוגן וכל ההפעלות של איבר הרקורסיה.

אם תריץ את אותו קוד ותספק כקלט את עובד 2 במקום עובד 5, תקבל את התוצאה מוצגת בטבלה 17–4.

טבלה 17–4: כפיפים של עובד 2

EmployeeID	*ReportsTo*	*FirstName*	*LastName*
2	NULL	Andrew	Fuller
1	2	Nancy	Davolio
3	2	Janet	Leverling
4	2	Margaret	Peacock
5	2	Steven	Buchanan
8	2	Laura	Callahan
6	5	Michael	Suyama
7	5	Robert	King
9	5	Anne	Dodsworth

כאן, איבר העוגן מחזיר את השורה של עובד 2. ההפעלה הראשונה של האיבר הרקורסיבי מחזירה כפיפים ישירים של עובד 2: עובדים 1, 3, 4, 5 ו–8. ההפעלה השנייה של האיבר הרקורסיבי מחזירה כפיפים ישירים של עובדים 1, 3, 4, 5 ו–8: עובדים 6, 7 ו–9. ההפעלה השלישית של האיבר הרקורסיבי מחזירה סט ריק והרקורסיה נעצרת. השאילתה החיצונית מחזירה את האיחוד של הסטים של התוצאה עם השורות עבור עובדים: 2, 1, 3, 4, 5, 8, 6, 7 ו–9.

אם אתה חושד שהנתונים שלך עלולים להכיל מחזורים מעגליים, תוכל להגדיר רמז MAXRECURSION (hint) כאמצעי ביטחון כדי להגביל את מספר ההפעלות של האיבר הרקורסיבי. הרמז מוגדר מייד לאחר השאילתה החיצונית:

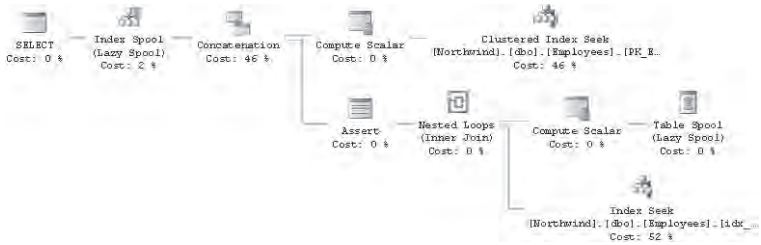
```
WITH cte_name AS (cte_body) outer_query OPTION(MAXRECURSION n);
```

בשורת קוד זו, n הוא הגבול למספר המחזורים הרקורסיביים. ברגע שהשאילתה מגיעה לגבול, היא נכשלת והודעת שגיאה מיוצרת. שים לב שברירת המחדל של MAXRECURSION היא 100. אם ברצונך להסיר מגבלה זו, הגדר MAXRECURSION 0. הגדרה כזו ניתנת למימוש רק ברמת השאילתה; לא קיימת אפשרות לשנות את ברירת המחדל ברמת ה–session, מסד הנתונים או השרת.

כדי להבין כיצד SQL Server מעבד CTE רקורסיבי, בחן את תוכנית העבודה בתרשים 4–4, אשר הופקה עבור השאילתה הקודמת המחזירה כפיפים של עובד 5.

כפי שתוכל לראות בתוכנית, סט התוצאה של איבר העוגן (השורה עבור עובד 5) מאוחזרת על ידי שימוש בפעולת חיפוש באינדקס–clustered (על הטור EmployeeID). האופרטור Compute Scalar מחשב מונה חזרות, המכוון בתחילה ל–0 (במופע הראשון של Compute Scalar בתוכנית) ומגדיל באחד בכל חזרה של האיבר הרקורסיבי (המופע השני של Compute Scalar בתוכנית).

תרשים 4–4: תוכנית עבודה עבור CTE רקורסיבי



כל סט זמני המופק הן על ידי איבר העוגן והן על ידי האיבר הרקורסיבי נאסף בטבלה זמנית חבויה (האופרטור Table Spool).

תוכל גם להבחין בהמשך התוכנית כי אינדקס זמני נוצר (מצוין על ידי האופרטור Index Spool). האינדקס נוצר על מונה החזרות בנוסף למאפיינים שאוחזרו (EmployeeID ,ReportsTo, FirstName, LastName).

הסט הזמני של כל הרצה של האיבר הרקורסיבי מאוחזר על ידי שימוש בפעולת חיפוש באינדקס הכיסוי שיצרתי עבור השאילתה. האופרטור Nested Loops מפעיל חיפוש לכל מנהל שהוחזר ונאסף בשלב הקודם, כדי להביא את הכפיפים הישירים שלו.

האופרטור Assert בודק האם מונה החזרות חרג מ–100 (גבול ברירת המחדל של MAXRECURSION). אופרטור זה הוא הממונה על עצירת השאילתה במקרה שמספר ההפעלות של האיבר הרקורסיבי חורג מהגבול של MAXRECURSION.

האופרטור Concatenation משרשר (מאחד) את כל הסטים הזמניים של התוצאות.

כשתסיים להתנסות עם ה–CTE הרקורסיבי, הסר את האינדקס שנוצר למטרה זו:

```
DROP INDEX dbo.Employees.idx_mgr_emp_ifname_ilname;
```

פונקציות דירוג אנליטיות

SQL Server 2005 מציג ארבע פונקציות דירוג אנליטיות חדשות (Analytical Ranking Functions): ROW_NUMBER, RANK, DENSE_RANK ו–NTILE. פונקציות אלו מספקות דרך פשוטה ויעילה ביותר להפקת חישובי דירוג.

ROW_NUMBER היא ללא ספק החידוש האהוב עלי ב– SQL Server 2005. על אף שעל פני השטח ייתכן שהוא אינו נראה חידוש כה משמעותי ביחס לחידושים אחרים (למשל, שאילתות רקורסיביות), יש לו כמות מדהימה של יישומים מעשיים המתפרשים הרבה מעבר לחישובים הקלאסיים של דירוג וניקוד. על ידי שימוש בפונקציה ROW_NUMBER הצלחתי לכוונן פתרונות רבים, כפי שאדגים לאורך כל הספר.

על אף שפונקציות הדירוג האחרות מחושבות טכנית מאחורי הקלעים בצורה דומה, יש להן הרבה פחות שימושים מעשיים. RANK ו– DENSE_RANK משמשים בעיקר למטרות דירוג וניקוד. NTILE משמש יותר למטרות ניתוח.

הצורך בחישובים כאלו היה קיים תמיד, אך השיטות להשגתם סבלו ממגבלה אחת או יותר – הן היו איטיות ביותר, מורכבות, או שלא לפי התקן. מכיוון שספר זה מיועד למשתמשים הן של SQL Server 2000 והן של 2005, אציג את השיטות המתאימות לכל גרסה. מכיוון שכל פונקציות הדירוג מחושבות טכנית מאחורי הקלעים בשיטות דומות, ברוב סעיף זה אטפל במספרי שורה ואדון בעיקר באספקטים הלוגיים של הפונקציות האחרות. לאחר שאכסה את השיטות לחישוב מספרי שורה, אציג בוחן ביצועים (benchmark) המשווה את ביצועיהם עם טבלאות בגדלים שונים. בדוגמאות שלי, אשתמש בטבלת Sales, אותה עליך ליצור ולמלא בנתונים על ידי הרצת הקוד בקטע–קוד 3–4.

קטע–קוד 3–4: יצירה ומילוי של טבלת Sales

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Sales') IS NOT NULL
  DROP TABLE dbo.Sales;
GO

CREATE TABLE dbo.Sales
(
  empid VARCHAR(10)  NOT NULL PRIMARY KEY,
  mgrid VARCHAR(10)  NOT NULL,
  qty   INT          NOT NULL
);

INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('A', 'Z', 300);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('B', 'X', 100);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('C', 'X', 200);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('D', 'Y', 200);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('E', 'Z', 250);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('F', 'Z', 300);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('G', 'X', 100);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('H', 'Y', 150);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('I', 'X', 250);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('J', 'Z', 100);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('K', 'Y', 200);

CREATE INDEX idx_qty_empid ON dbo.Sales(qty, empid);
CREATE INDEX idx_mgrid_qty_empid ON dbo.Sales(mgrid, qty, empid);
```

התוכן של הטבלה Sales המוחזרת על ידי השאילתה הבאה מוצג בטבלה 18–4:

```
SELECT * FROM dbo.Sales;
```

טבלה 18–4: תוכן של הטבלה Sales

empid	*mgrid*	*qty*
A	Z	300
B	X	100
C	X	200
D	Y	200
E	Z	250
F	Z	300
G	X	100
H	Y	150
I	X	250
J	Z	100
K	Y	200

פונקציות הדירוג של SQL Server 2005 יכולות להופיע בשאילתה רק בפסוקיות SELECT ו– OREDR BY. המבנה הכללי של פונקציית דירוג הוא כלהלן:

```
ranking_function OVER([PARTITION BY col_list] ORDER BY col_list)
```

הפסוקית האופציונלית PARTITION BY מאפשרת לבקש שערכי הדירוג יחושבו לכל מחיצה (או קבוצה) של שורות בנפרד. למשל, אם תגדיר mgrid בפסוקית PARTITION BY, ערכי הדירוג יחושבו לשורות של כל מנהל בנפרד. בפסוקית ORDER BY, אתה מציין את רשימת הטורים הקובעים את סדר ההקצאה של ערכי הדירוג.

האינדקס האופטימלי לחישובי דירוג (ללא תלות בשיטה בה אתה משתמש) הוא זה שנוצר על partitioning_columns, sort_columns, covered_cols. יצרתי אינדקסים אופטימליים על טבלת Sales עבור מספר בקשות חישוב דירוג.

מספר שורה

מספרי שורה הם רצף של מספרים שלמים המוקצים לשורות של סט תוצאה של שאילתה בהתבסס על מיון מסוים. בסעיפים הבאים אתאר את הכלים ואת השיטות לחישוב מספרי שורה הן ב– SQL Server 2005 והן בגרסאות קודמות.

הפונקציה ROW_NUMBER ב- SQL Server 2005

הפונקציה ROW_NUMBER מקצה רצף של מספרים שלמים לשורות של סט תוצאה של שאילתה בהתבסס על מיון מסוים, אפשר גם שבתוך מחיצה (partition). למשל, השאילתה הבאה (המייצרת את הפלט המוצג בטבלה 19–4) מחזירה שורות מכירה של עובד ומקצה מספרי שורה לפי מיון qty:

```
SELECT empid, qty,
  ROW_NUMBER() OVER(ORDER BY qty) AS rownum
FROM dbo.Sales
ORDER BY qty;
```

טבלה 19–4: מספרי שורה לפי מיון qty

empid	*qty*	*rownum*
B	100	1
G	100	2
J	100	3
H	150	4
K	200	5
C	200	6
D	200	7
E	250	8
I	250	9
F	300	10
A	300	11

כדי להבין את היעילות של פונקציות הדירוג ב– SQL Server 2005, בחן את תוכנית העבודה המוצגת בתרשים 5–4, שנוצרה עבור שאילתה זו.

תרשים 5–4: תוכנית עבודה עבור ROW_NUMBER

כדי לחשב ערכי דירוג, ה–optimizer צריך שהנתונים ימוינו ראשית על פי טור או טורי החיץ (partitioning) ואז על פי טור או טורי המיון.

אם קיים כבר אינדקס המתחזק את הנתונים בסדר המבוקש, רמת העלה של האינדקס נסרקת פשוט לפי הסדר (כמו בדוגמה שלנו). אחרת, הנתונים ייסרקו ואז ימוינו בעזרת אופרטור מיון. האופרטור Sequence Project הוא האופרטור הממונה על חישוב ערכי הדירוג. לכל שורת קלט, הוא זקוק לשני "דגלים":

1. האם השורה היא הראשונה במחיצה? אם כן, האופרטור Sequence Project יאפס את ערך הדירוג.
2. האם ערך הדירוג בשורה זו שונה מזה שבשורה הקודמת? אם כן, האופרטור Sequence Project יגדיל את ערך הדירוג כפי שמוכתב על ידי פונקציית הדירוג הספציפית.

לכל פונקציות הדירוג, האופרטור Segment ייצר את הערך של הדגל הראשון.

האופרטור Segment למעשה קובע את גבולות הקיבוץ. הוא שומר שורה אחת בזיכרון ומשווה אותה עם הבאה. אם הן שונות, הוא מנפק ערך אחד. אם הן זהות, הוא מנפק ערך אחר.

כדי לייצר את הדגל הראשון, המציין האם השורה היא הראשונה במחיצה, האופרטור Segment משווה את ערכי הטור של ה– PARTITION BY בין השורות הנוכחיות לקודמות. כמובן שהוא מנפק "true" עבור השורה הראשונה שנקראת. מהשורה השנייה ואילך, הפלט שלו תלוי בשאלה האם ערך הטור של ה– PARTITION BY השתנה. בדוגמה שלנו, לא הגדרתי פסוקית PARTITION BY, כך שכל הטבלה מתנהגת כמחיצה אחת. במקרה זה, Segment ינפק "true" עבור השורה הראשונה ו–"false" לכל האחרות.

באשר לדגל השני (העונה לשאלה "האם הערך שונה מהערך הקודם?"), האופרטור שייחשב זאת תלוי בשאלה איזו פונקציית דירוג ביקשת. עבור ROW_NUMBER, ערך הדירוג חייב לגדול עבור כל שורה ללא תלות בשינוי של ערך המיון. כך שבמקרה שלנו אופרטור Compute Scalar פשוט מנפק "true" (1) בכל פעם. במקרים אחרים (למשל, עם פונקציות RANK ו– DENSE_RANK), אופרטור Segment אחר ישמש כדי לומר לאופרטור Sequence Project האם ערך המיון השתנה כדי לקבוע האם להגדיל את ערך הדירוג או לא.

ייתכן שהחוכמה של תוכנית זו ושל השיטות בהן משתמש ה–optimizer לחישוב ערכי דירוג, עדיין אינה נראית לעין. נכון לעכשיו מספיק לומר שהנתונים נסרקים פעם אחת בלבד, ואם אינם ממוינים כבר בתוך האינדקס, הרי שיידרש גם מיון מפורש. זוהי שיטה מהירה בהרבה מכל שיטה שהייתה קיימת לחישוב ערכי דירוג ב– SQL Server 2000, כפי שאדגים בפירוט מייד.

דטרמיניזם

כפי שבוודאי הבחנת בפלט של השאילתה הקודמת, מספרי שורה ממשיכים לגדול ללא תלות בשינוי של ערך המיון. מספרי שורה בתוך מחיצה חייבים להיות ייחודיים. המשמעות

היא, שעבור רשימה שאינה ממוינת ייחודית, השאילתה אינה דטרמיניסטית. כלומר, ישנם סטים של תוצאה שונים שהם נכונים ולא רק אחד. למשל, בטבלה 19–4 ניתן לראות שעובדים B, G ו–J, שלכולם כמות 100, קיבלו את מספרי השורה 1, 2 ו–3, בהתאמה. אך התוצאה תהיה תקפה גם אם שלושת עובדים אלו היו מקבלים מספרי שורה 1, 2 ו–3 בסדר שונה.

ישנם יישומים המחייבים דטרמיניזם. כדי להבטיח דטרמיניזם, כל שצריך לעשות זה להוסיף שובר–שוויון הגורם לערכים של partitioning column(s) + sort column(s) להיות ייחודיים.

למשל, השאילתה הבאה (המייצרת את הפלט המוצג בטבלה 20–4) מדגימה הן מספר שורה לא–דטרמיניסטי המתבסס על הטור qty בלבד והן מספר שורה דטרמיניסטי המתבסס על המיון של qty ושל empid:

```
SELECT empid, qty,
  ROW_NUMBER() OVER(ORDER BY qty)          AS nd_rownum,
  ROW_NUMBER() OVER(ORDER BY qty, empid)   AS d_rownum
FROM dbo.Sales
ORDER BY qty, empid;
```

טבלה 20–4: מספרי שורה, דטרמיניזם

empid	*qty*	*nd_rownum*	*d_rownum*
B	100	1	1
G	100	2	2
J	100	3	3
H	150	4	4
C	200	6	5
D	200	7	6
K	200	5	7
E	250	8	8
I	250	9	9
A	300	11	10
F	300	10	11

חציצה (partitioning)

כפי שהזכרתי קודם, ניתן לחשב ערכי דירוג גם בתוך מחיצות (קבוצות של שורות). הדוגמה הבאה (המייצרת את הפלט המוצג בטבלה 21–4) מחשבת מספרי שורה בהתבסס על המיון של qty ושל empid, לכל מנהל בנפרד:

```
SELECT mgrid, empid, qty,
  ROW_NUMBER() OVER(PARTITION BY mgrid ORDER BY qty, empid) AS rownum
FROM dbo.Sales
ORDER BY mgrid, qty, empid;
```

טבלה 21–4: מספרי שורה, מחיצות

mgrid	*empid*	*qty*	*rownum*
X	B	100	1
X	G	100	2
X	C	200	3
X	I	250	4
Y	H	150	1
Y	D	200	2
Y	K	200	3
Z	J	100	1
Z	E	250	2
Z	A	300	3
Z	F	300	4

שיטה מבוססת–סטים לפני SQL Server 2005

קיימות מספר שיטות לחישוב ערכי דירוג ב– SQL Server בגרסאות קודמות ל– SQL Server 2005, וכולן סובלות ממגבלה כלשהי. בטרם אתחיל לתאר שיטות אלו, זכור שניתן לחשב ערכי דירוג גם אצל הלקוח. בכל דרך שתבחר, הלקוח יסרוק את הרשומות המוחזרות מ– SQL Server אחת אחר השנייה. הלקוח יכול פשוט לבקש את השורות ממוינות, ובתוך לולאה, להגדיל את המונה. כמובן שאם אתה זקוק לערכי הדירוג למניפולציה נוספת בצד–השרת בטרם נשלחות התוצאות ללקוח, הרי שלא קיימת אפשרות של דירוג בצד–הלקוח.

אתחיל בשיטה הרגילה מבוססת–הסטים. לרוע המזל, זו בדרך כלל האיטית ביותר.

טור מיון ייחודי

לפני SQL Server 2005, היו אפשריים חישובים של מספרי שורה מבוססי–סטים פשוטים יחסית, בהינתן שילוב ייחודי של partitioning + sort column(s). כפי שאתאר בהמשך, קיימים גם חישובים של מספרי שורה מבוססי–סטים ללא שילוב ייחודי זה, אך הם מורכבים יותר משמעותית.

כל חישובי ערכי דירוג ניתנים להשגה על ידי ספירת שורות. כדי לחשב מספרי שורה, ניתן להשתמש בשיטה הבסיסית הבאה. כל שעליך לעשות זה להשתמש בתת–שאילתה כדי לספור את מספר השורות עם ערך מיון קטן או שווה. ספירה זו מתייחסת למספר השורה המבוקש. למשל, השאילתה הבאה מייצרת מספרי שורה בהתבסס על empid, ומפיקה את הפלט בטבלה 22–4:

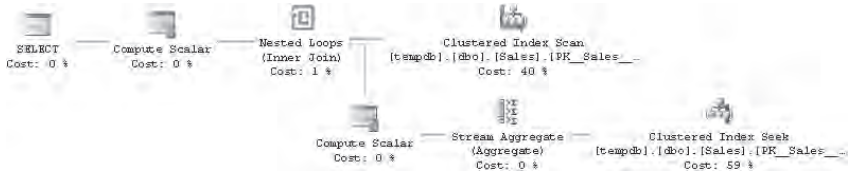
```
SELECT empid,
  (SELECT COUNT(*)
   FROM dbo.Sales AS S2
   WHERE S2.empid <= S1.empid) AS rownum
FROM dbo.Sales AS S1
ORDER BY empid;
```

טבלה 22–4: מספרי שורה, טור מיון ייחודי

empid	*rownum*
A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9
J	10
K	11

שיטה זו לחישוב מספרי שורה, על אף שפשוטה יחסית, איטית ביותר. כדי להבין מדוע, בחן את תוכנית העבודה המוצגת בתרשים 6–4 שנוצרה עבור השאילתה.

תרשים 6–4: תוכנית עבודה עבור שאילתת מספרי שורה מבוססת–סטים, טרום SQL Server 2005



על טור המיון (empid) קיים אינדקס שהוא במקרה גם האינדקס–Clustered של טבלת Sales. ראשית הטבלה נסרקת במלואה (אופרטור Clustered Index Scan) כדי להחזיר את כל השורות. לכל שורה המוחזרת מהסריקה המלאה הראשונית, האופרטור Nested Loops מפעיל את הפעילות המפיקה את מספר השורה על ידי ספירת שורות. כל חישוב של מספר שורה מערב פעולת חיפוש בתוך אינדקס ה–clustered, שלאחריה פעולת סריקה חלקית (מראש הרשימה המקושרת של רמת העלה לנקודה האחרונה שבה inner_empid קטן או שווה ל–outer_empid).

שים לב שקיימים שני אופרטורים שונים המשתמשים באינדקס ה–clustered – ראשית, סריקה מלאה להחזרת כל השורות; שנית, חיפוש שלאחריו סריקה חלקית לכל שורה חיצונית כדי להשיג את הספירה.

זכור שהגורם העיקרי המשפיע על ביצועים של שאילתות המבצעות מניפולציה בנתונים יהיה לרוב I/O. הערכה גסה של מספר השורות אליהן ניגשים תראה עד כמה תוכנית עבודה זו אינה יעילה. כדי לחשב מספר שורה עבור השורה הראשונה בטבלה, על SQL Server לסרוק שורה אחת באינדקס. עבור השורה השנייה, עליו לסרוק 2 שורות. עבור השורה השלישית, עליו לסרוק 3 שורות, וכך הלאה, ועבור שורה n בטבלה עליו לסרוק n שורות. עבור טבלה בת n שורות, בהתקיים אינדקס המבוסס על טור המיון, המספר הכולל של שורות שייסרקו הוא 1+2+3+...+n. ייתכן שאינך מבין מיידית עד כמה מספר השורות שייסרקו הוא גדול. כדי לתת לך תחושה, עבור טבלה בעלת 100,000 שורות, אתה צפוי לסך של 5,000,050,000 שורות שייסרקו.

כהערת אגב, יש סיפור על המתמטיקאי גאוס. כשהיה ילד, הוא ובני כיתתו קיבלו משימה מהמורה שלהם, למצוא את הסכום של כל השלמים מ–1 עד 100. גאוס סיפק את התשובה כמעט מיידית. כשהמורה שאל אותו כיצד הגיע לתשובה כה מהר, הוא ענה שחיבר את הערכים הראשון והאחרון (1 + 100 = 101), ואז הכפיל את התוצאה במחצית ממספר השלמים (50), שהוא מספר הזוגות. ואכן, התוצאה של **ערך_ראשון + ערך_אחרון** שווה לתוצאה של **ערך_שני + ערך_לפני_אחרון** וכך הלאה. בקצרה, הנוסחה לחישוב הסכום של n השלמים החיוביים הראשונים היא $(n + n^2)/2$. זהו סך השורות שיש לסרוק כדי לחשב מספרי שורה בהשתמש בשיטה זו כאשר קיים אינדקס המבוסס על טור המיון. אתה צפוי לגרף n^2 של עלות I/O וזמן ריצה המתבסס על מספר השורות בטבלה. תוכל לשחק עם המספרים בנוסחה ולראות שהעלות מגיעה לממדים מפלצתיים מהר למדי.

אם לא קיים אינדקס על הטבלה, מצב העניינים אפילו מחמיר. כדי לחשב כל מספר שורה, יש לסרוק את כל הטבלה. המספר הכולל של שורות שייסרקו על ידי השאילתה הוא אז n^2. למשל, בהינתן טבלה של 100,000 שורות, השאילתה תסרוק סך של 10,000,000,000 שורות.

טור מיון לא ייחודי ושובר–שוויון

כאשר טור המיון אינו ייחודי, ניתן להפוך אותו לייחודי על ידי הצגת שובר–שוויון, כדי לאפשר פתרון ששומר על רמה סבירה של פשטות. הנח שטור המיון הוא sortcol, וטור שובר–השוויון הוא tiebreaker. כדי לספור שורות עם ערכים זהים או קטנים יותר של רשימת המיון (sortcol+tiebreaker), השתמש בתת–השאילתה בביטוי הבא:

```
inner_sortcol < outer_sortcol
OR inner_sortcol = outer_sortcol
   AND inner_tiebreaker <= outer_tiebreaker
```

שים לב שסדר הקדימות של האופרטורים מכתיב ש–AND יוערך לפני OR. שקול שימוש בסוגריים לצורך בהירות, קריאות וקלות–תחזוקה.

השאילתה הבאה (המייצרת את הפלט המוצג בטבלה 23–4) מייצרת מספרי שורה בהתבסס על qty ו–empid, בסדר זה:

```
SELECT empid, qty,
  (SELECT COUNT(*)
   FROM dbo.Sales AS S2
   WHERE S2.qty < S1.qty
      OR (S2.qty = S1.qty AND S2.empid <= S1.empid)) AS rownum
FROM dbo.Sales AS S1
ORDER BY qty, empid;
```

טבלה 23–4: מספרי שורה, טור מיון לא ייחודי ושובר–שוויון

empid	*qty*	*rownum*
B	100	1
G	100	2
J	100	3
H	150	4
C	200	5
D	200	6

empid	*qty*	*rownum*
K	200	7
E	250	8
I	250	9
A	300	10
F	300	11

טור מיון לא ייחודי ללא שובר–שוויון

הבעיה הופכת להיות מורכבת בצורה משמעותית כאשר עליך להקצות מספרי שורה לפי טור מיון לא ייחודי ללא שימוש בשובר–שוויון, בשיטה מבוססת–סטים טרום SQL Server 2005. למשל, בהינתן הטבלה T1, שתצור ותמלא על ידי הקוד בקטע–קוד 4–4, נניח שעליך לייצר מספרי שורה בהתבסס על מיון col1.

קטע–קוד 4–4: יצירה ומילוי של טבלה T1

```
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO
CREATE TABLE dbo.T1(col1 VARCHAR(5));
INSERT INTO dbo.T1(col1)  VALUES('A');
INSERT INTO dbo.T1(col1)  VALUES('A');
INSERT INTO dbo.T1(col1)  VALUES('A');
INSERT INTO dbo.T1(col1)  VALUES('B');
INSERT INTO dbo.T1(col1)  VALUES('B');
INSERT INTO dbo.T1(col1)  VALUES('C');
INSERT INTO dbo.T1(col1)  VALUES('C');
INSERT INTO dbo.T1(col1)  VALUES('C');
INSERT INTO dbo.T1(col1)  VALUES('C');
INSERT INTO dbo.T1(col1)  VALUES('C');
```

הפתרון חייב להיות תואם SQL Server 2000, כך שאין באפשרותך פשוט להשתמש בפונקציה ROW_NUMBER. כמו כן, הפתרון חייב להתאים לתקן.

בפתרון לבעיה זו, אשתמש ראשית בשיטת–מפתח חשובה ביותר – יצירת שורות כפולות תוך שימוש בטבלת עזר של מספרים. ליתר דיוק, השיטה מייצרת עותקים ממוספרים ברצף של כל שורה, אך לשם הפשטות, אתייחס לכך לאורך הספר כ"יצירת כפילויות".

את מושג טבלת העזר של מספרים וכיצד לייצר כזו אסביר בהמשך הפרק בסעיף "טבלת עזר של מספרים". לעת עתה, הרץ פשוט את הקוד בקטע–קוד 8–4, המייצר את טבלת Nums וממלא אותה ב–1,000,000 השלמים שבתחום 1 <= n <= 1,000,000.

השיטה בה אדון היא "יצירת כפילויות" או "הרחבת" מספר השורות. למשל, בהינתן הטבלה T1, נניח שברצונך לייצר 5 עותקים של כל שורה. כדי להשיג זאת, באפשרותך להשתמש בטבלה Nums כלהלן:

```
SELECT ... FROM dbo.T1, dbo.Nums WHERE n <= 5;
```

בפרק 5 אספק מידע נוסף על השיטה "יצירת כפילויות" ושימושיה.

נחזור לבעיה המקורית, עליך לייצר מספרי שורה לשורות ב–T1, בהתבסס על מיון col1. הצעד הראשון בפתרון הוא "קריסת" השורות על ידי קיבוצם לפי col1. לכל קבוצה, אתה מחזיר את מספר הכפילויות (ספירה של השורות בקבוצה). אתה מחזיר גם, על ידי שימוש בתת–שאילתה, את מספר השורות בטבלת הבסיס שלהן ערך מיון קטן יותר. להלן השאילתה המבצעת את הצעד הראשון, והפלט שלה מוצג בטבלה 24–4:

```
SELECT col1, COUNT(*) AS dups,
  (SELECT COUNT(*) FROM dbo.T1 AS B
   WHERE B.col1 < A.col1) AS smaller
FROM dbo.T1 AS A
GROUP BY col1;
```

טבלה 24–4: מספרי שורה, טור מיון לא ייחודי, ללא שובר–שוויון, פלט של צעד 1

col1	*dups*	*smaller*
A	3	0
B	2	3
C	5	5

לדוגמה, A מופיע 3 פעמים, וקיימות 0 שורות עם ערך col1 קטן מ–A. B מופיע פעמיים, וקיימות 3 שורות עם ערך col1 קטן מ–B. וכך הלאה.

הצעד הבא (המפיק את הפלט המופיע בטבלה 25–4) הוא להרחיב את מספר השורות או ליצור עותקים ממוספרים ברצף של כל שורה. צעד זה מושג על ידי יצירת טבלה נגזרת מתוך השאילתה הקודמת ואיחודה לטבלה Nums כלהלן, בהתבסס על n <= dups:

```
SELECT col1, dups, smaller, n
FROM (SELECT col1, COUNT(*) AS dups,
        (SELECT COUNT(*) FROM dbo.T1 AS B
         WHERE B.col1 < A.col1) AS smaller
      FROM dbo.T1 AS A
      GROUP BY col1) AS D, Nums
WHERE n <= dups;
```

טבלה 4–25: מספרי שורה, טור מיון לא ייחודי, ללא שובר–שוויון, פלט צעד 2

col1	*dups*	*smaller*	*n*
A	3	0	1
A	3	0	2
A	3	0	3
B	2	3	1
B	2	3	2
C	5	5	1
C	5	5	2
C	5	5	3
C	5	5	4
C	5	5	5

כעת בחן היטב את הפלט בטבלה 4–25, וראה האם תוכל לנחש כיצד לייצר את מספרי השורה.

מספר השורה יכול להיות מבוטא כמספר השורות עם ערך מיון קטן יותר, בתוספת מספר השורה בתוך אותה קבוצת ערך מיון – במילים אחרות, n + smaller. השאילתה הבאה היא הפתרון הסופי, והיא מייצרת את הפלט המוצג בטבלה 4–26:

```
SELECT n + smaller AS rownum, col1
FROM (SELECT col1, COUNT(*) AS dups,
        (SELECT COUNT(*) FROM dbo.T1 AS B
         WHERE B.col1 < A.col1) AS smaller
      FROM dbo.T1 AS A
      GROUP BY col1) AS D, Nums
WHERE n <= dups;
```

טבלה 4–26: מספרי שורה, טור מיון לא ייחודי, ללא שובר–שוויון, פלט סופי

rownum	*col1*
1	A
2	A
3	A
4	B
5	B

rownum	col1
6	C
7	C
8	C
9	C
10	C

חציצה

חציצה מושגת פשוט על ידי הוספת קישור (correlation) בתת–השאילתה בהתבסס על התאמה בין טור או טורי החציצה בטבלאות הפנימית והחיצונית. למשל, השאילתה הבאה על טבלת Orders (המייצרת את הפלט המופיע בטבלה 27–4) מחשבת מספרי שורה המקובצים על ידי mgrid, ממוינים על ידי qty, ומשתמשים ב–empid כשובר–שוויון:

```
SELECT mgrid, empid, qty,
  (SELECT COUNT(*)
   FROM dbo.Sales AS S2
   WHERE S2.mgrid = S1.mgrid
      AND (S2.qty < S1.qty
           OR (S2.qty = S1.qty AND S2.empid <= S1.empid))) AS rownum
FROM dbo.Sales AS S1
ORDER BY mgrid, qty, empid;
```

טבלה 27–4: מספרי שורה, במחיצות

mgrid	empid	qty	rownum
X	B	100	1
X	G	100	2
X	C	200	3
X	I	250	4
Y	H	150	1
Y	D	200	2
Y	K	200	3
Z	J	100	1
Z	E	250	2
Z	A	300	3
Z	F	300	4

שים לב: כפי שציינתי קודם לכן, לשיטה מבוססת–הסטים טרום SQL Server 2005 לחישוב מספרי שורה יש עלות של num_rows^2. יחד עם זאת, למספר קטן יחסית של שורות (באזור העשרות), היא מהירה למדי. שים לב שבעיית הביצועים קשורה יותר לגודל המחיצה מאשר לגודל הטבלה. אם תיצור את האינדקס המומלץ בהתבסס על partitioning_cols, sort_cols, tiebreaker_cols, מספר שורות שייסרקו בתוך האינדקס שווה למספר השורות שנוצרו. מספר השורה מתאפס (מתחיל מ–1) בכל מחיצה חדשה. כך שאפילו עבור טבלאות גדולות מאוד, כאשר גודל המחיצה הוא יחסית קטן וברשותך אינדקס טוב, הפתרון מהיר למדי. אם קיימות p מחיצות ו–r שורות בכל מחיצה, סך השורות שייסרקו הוא: $p * (r + r^2) / 2$. למשל, אם ברשותך 100,000 מחיצות ו–10 שורות בכל מחיצה, סך הכל ייסרקו 5,500,000. אף על פי שהמספר נראה גדול, הוא אפילו אינו מתקרב למספר אליו תגיע ללא חציצה. וכל עוד גודל המחיצה נשאר קבוע, הגרף של עלות השאילתה בהשוואה למספר השורות בטבלה הוא ליניארי.

פתרון מבוסס–סמן

ניתן להשתמש בסמן כדי לחשב מספרי שורה. לעומת החלופות שהוזכרו קודם, פתרון מבוסס–סמן הוא פשוט למדי. אתה יוצר סמן FAST_FORWARD (קריאה–בלבד, קדימה–בלבד) בהתבסס על שאילתה הממיינת את הנתונים לפי partitioning_cols, sort_cols, tiebreaker_cols. תוך כדי הבאת רשומות מהסמן, אתה פשוט מגדיל מונה, ומאפס אותו בכל פעם שאתה מזהה מחיצה חדשה. באפשרותך לאחסן את שורות התוצאה בצירוף מספרי השורה בטבלה זמנית או במשתנה טבלה.

לשם הדוגמה, הקוד בקטע–קוד 5–4 (המייצר את הפלט המוצג בטבלה 28–4) משתמש בסמן כדי לחשב מספרי שורה בהתבסס על המיון של qty ו–empid:

קטע–קוד 5–4: חישוב מספרי שורה עם סמן

```
DECLARE @SalesRN TABLE(empid VARCHAR(5), qty INT, rn INT);
DECLARE @empid AS VARCHAR(5), @qty AS INT, @rn AS INT;

BEGIN TRAN

DECLARE rncursor CURSOR FAST_FORWARD FOR
   SELECT empid, qty FROM dbo.Sales ORDER BY qty, empid;
OPEN rncursor;

SET @rn = 0;
```

```
FETCH NEXT FROM rncursor INTO @empid, @qty;
WHILE @@fetch_status = 0
BEGIN
   SET @rn = @rn + 1;
   INSERT INTO @SalesRN(empid, qty, rn) VALUES(@empid, @qty, @rn);
   FETCH NEXT FROM rncursor INTO @empid, @qty;
END

CLOSE rncursor;
DEALLOCATE rncursor;

COMMIT TRAN

SELECT empid, qty, rn FROM @SalesRN;
```

טבלה 28–4: פלט של סמן המחשב מספרי שורה

empid	*qty*	*rn*
B	100	1
G	100	2
J	100	3
H	150	4
C	200	5
D	200	6
K	200	7
E	250	8
I	250	9
A	300	10
F	300	11

באופן כללי, יש להימנע מעבודה עם סמנים, שכן לסמנים יש תקורה רבה המכבידה על הביצועים. אף על פי כן, במקרה זה, אלא אם כן גודל המחיצה זעיר, לפתרון מבוסס–הסמן ביצועים טובים הרבה יותר מאשר השיטה מבוססת–הסטים טרום SQL Server 2005, שכן הוא סורק את הנתונים פעם אחת בלבד. המשמעות היא שככל שהטבלה גדלה, רמת הביצועים של הפתרון מבוסס–הסמן מידרדרת ליניארית, בניגוד להידרדרות של n^2 בפתרון מבוסס–הסטים טרום SQL Server 2005.

פתרון מבוסס–IDENTITY

בגרסאות קודמות ל– SQL Server 2005 ניתן להשתמש בפונקציה IDENTITY ובמאפיין–הטור IDENTITY כדי לחשב מספרי שורה. לפיכך, חישוב מספרי שורה עם IDENTITY היא שיטה שכדאי להכיר. אך בטרם נמשיך, עליך לדעת שכאשר אתה משתמש בפונקציה IDENTITY, אינך יכול להבטיח את סדר ההקצאה של ערכי IDENTITY. אף על פי כן, באפשרותך להבטיח את סדר ההקצאה על ידי שימוש בטור IDENTITY במקום בפונקציה IDENTITY: ראשית צור טבלה בעלת טור IDENTITY ואז הכנס את נתונים על ידי שימוש במשפט INSERT SELECT עם פסוקית ORDER BY.

מידע נוסף: אני ממליץ בחום על קריאת מאמר Knowledge Base 273586 (*http://support.com/default.aspx?acid=kb;en-us;273586*), שם ניתן למצוא דיון מפורט בנושא IDENTITY ו- OREDER BY.

ללא–מחיצות

שימוש בפונקציה IDENTITY במשפט SELECT INTO הוא ללא ספק הדרך המהירה ביותר לחישוב מספרי שורה בשרת טרום SQL Server 2005. הסיבה הראשונה לכך היא שהנתונים נסרקים פעם אחת בלבד, ללא התקורה המועמסת במניפולציה של סמן. הסיבה השנייה היא ש– SELECT INTO היא פעולה עם רישום מינימלי ב– transaction log כאשר מודל ה–RECOVERY של מסד הנתונים אינו FULL. עם זאת, זכור שניתן לסמוך על פעולה זו רק אם סדר ההקצאה של מספרי השורה אינו חשוב.

למשל, הקוד הבא מדגים כיצד להשתמש בפונקציה IDENTITY ליצירת טבלה זמנית והכנסת מספרי שורה, שלא לפי סדר מסוים:

```
SELECT empid, qty, IDENTITY(int, 1, 1) AS rn
INTO #SalesRN FROM dbo.Sales;

SELECT * FROM #SalesRN;

DROP TABLE #SalesRN;
```

טכניקה זו שימושית כאשר אתה נדרש לייצר מספרים שלמים כאמצעי זיהוי להבחנה בין שורות.

אל תיתן לעובדה שניתן טכנית להגדיר פסוקית ORDER BY בשאילתה SELECT INTO להטעות אותך. ב– SQL Server 2000, אין ערובה לכך שבתוכנית העבודה הקצאת ערכי IDENTITY תקרה לאחר המיון. ב– SQL Server 2005 אין לכך משמעות מכיוון שבכל מקרה תשתמש בפונקציה ROW_NUMBER.

כפי שהזכרתי קודם, כאשר סדר הקצאת ערכי IDENTITY כן חשוב – במילים אחרות, כאשר מספרי השורה צריכים להתבסס על סדר נתון – ראשית צור את הטבלה, ואז הכנס את הנתונים. שיטה זו אינה מהירה כגישת ה– SELECT INTO מכיוון ש– INSERT INTO תמיד נרשמת ב– transaction log באופן מלא; אף על פי כן, היא עדיין מהירה הרבה יותר מהשיטות האחרות קודם ל– SQL Server 2005.

להלן דוגמה לחישוב מספרי שורה בהתבסס על המיון של qty ו–empid:

```
CREATE TABLE #SalesRN(empid VARCHAR(5), qty INT, rn INT IDENTITY);

INSERT INTO #SalesRN(empid, qty)
   SELECT empid, qty FROM dbo.Sales ORDER BY qty, empid;

SELECT * FROM #SalesRN;

DROP TABLE #SalesRN;
```

עם–מחיצות

שימוש בגישת IDENTITY ליצירת מספרי שורה בתוך מחיצות דורש צעד נוסף. כמו בפתרון ללא–מחיצות, הנתונים מוכנסים לתוך טבלה עם טור IDENTITY, אלא שהפעם הם ממוינים לפי partitioning_cols, sort_cols, tiebreaker_cols.

הצעד הנוסף הוא שאילתה המחשבת את מספרי השורה בתוך המחיצה על ידי שימוש בנוסחה הבאה: general_row_number – min_row_number_within_partition + 1. ניתן לקבל את מספר השורה הקטן ביותר בתוך המחיצה על ידי תת–שאילתה מקושרת או על ידי join.

לשם הדוגמה, הקוד בקטע–קוד 6–4 מייצר מספרי שורה מחולקים לפי mgrid וממוינים לפי qty ו–empid. הקוד מציג הן את גישת תת–השאילתה והן את גישת ה–join להשגת מספר השורה הקטן ביותר בתוך מחיצה.

קטע–קוד 6–4: חישוב מספרי שורה במחיצות עם IDENTITY

```
CREATE TABLE #SalesRN
   (mgrid VARCHAR(5), empid VARCHAR(5), qty INT, rn INT IDENTITY);
CREATE UNIQUE CLUSTERED INDEX idx_mgrid_rn ON #SalesRN(mgrid, rn);

INSERT INTO #SalesRN(mgrid, empid, qty)
   SELECT mgrid, empid, qty FROM dbo.Sales ORDER BY mgrid, qty, empid;
```

```
-- Option 1 – using a subquery
SELECT mgrid, empid, qty,
   rn - (SELECT MIN(rn) FROM #SalesRN AS S2
         WHERE S2.mgrid = S1.mgrid) + 1 AS rn
FROM #SalesRN AS S1;

-- Option 2 – using a join
SELECT S.mgrid, empid, qty, rn - minrn + 1 AS rn
FROM #SalesRN AS S
   JOIN (SELECT mgrid, MIN(rn) AS minrn
         FROM #SalesRN
         GROUP BY mgrid) AS M
      ON S.mgrid = M.mgrid;

DROP TABLE #SalesRN;
```

בוחן ביצועים

הצגתי ארבע שיטות שונות לחישוב מספרי שורה בצד–השרת. אחת קיימת רק ב– SQL Server 2005 (שימוש בפונקציה ROW_NUMBER), ושלוש (מבוססת–סטים תוך שימוש בתת–שאילתות, מבוססת–סמן, ומבוססת–IDENTITY) נתמכות בשתי הגרסאות. כפי שהזכרתי קודם, שלושת חישובי הדירוג שאתאר בהמשך הפרק מחושבים טכנית על ידי שימוש בשיטות גישה דומות מאוד, כך שהיבטי הביצועים בהם דנתי ובוחן–הביצועים שאציג להלן רלוונטיים לכל חישובי הדירוג.

הרצתי את בוחן–הביצועים המוצג בקטע–קוד 7–4 על המחשב שלי (CPU יחיד: Intel Centrino 1.7 Mhz; RAM: 1GB, כונן דיסק יחיד). על אף שהמחשב שלי הוא לא בדיוק המודל המשקף ביותר לשרת תפעולי, ניתן לקבל תחושה של הבדלי הביצועים בין השיטות. בוחן–הביצועים מכניס לטבלה מספר גדל והולך של שורות, תחילה 10,000 ומתקדם עד 100,000 בקפיצות של 10,000 שורות. בוחן–הביצועים מחשב מספרי שורה על ידי שימוש בכל ארבע השיטות, כאשר האפשרות Discard Results ב– SSMS (SQL Server Management Studio) מופעלת כדי להסיר את ההשפעה של יצירת פלט. בוחן–הביצועים מציין את זמני הריצה באלפיות–שנייה בטבלה RNBenchmark.

קטע–קוד 7–4: בוחן–ביצועים להשוואת שיטות לחישוב מספרי שורה

```
-- Change Tool's Options to Discard Query Results
SET NOCOUNT ON;
USE tempdb;
GO
```

```
IF OBJECT_ID('dbo.RNBenchmark') IS NOT NULL
  DROP TABLE dbo.RNBenchmark;
GO
IF OBJECT_ID('dbo.RNTechniques') IS NOT NULL
  DROP TABLE dbo.RNTechniques;
GO
IF OBJECT_ID('dbo.SalesBM') IS NOT NULL
  DROP TABLE dbo.SalesBM;
GO
IF OBJECT_ID('dbo.SalesBMIdentity') IS NOT NULL
  DROP TABLE dbo.SalesBMIdentity;
GO
IF OBJECT_ID('dbo.SalesBMCursor') IS NOT NULL
  DROP TABLE dbo.SalesBMCursor;
GO

CREATE TABLE dbo.RNTechniques
(
  tid INT NOT NULL PRIMARY KEY,
  technique VARCHAR(25) NOT NULL
);
INSERT INTO RNTechniques(tid, technique) VALUES(1, 'Set-Based 2000');
INSERT INTO RNTechniques(tid, technique) VALUES(2, 'IDENTITY');
INSERT INTO RNTechniques(tid, technique) VALUES(3, 'Cursor');
INSERT INTO RNTechniques(tid, technique) VALUES(4, 'ROW_NUMBER 2005');
GO

CREATE TABLE dbo.RNBenchmark
(
  tid         INT    NOT NULL REFERENCES dbo.RNTechniques(tid),
  numrows     INT    NOT NULL,
  runtimems   BIGINT NOT NULL,
  PRIMARY KEY(tid, numrows)
);
GO

CREATE TABLE dbo.SalesBM
(
  empidINT NOT NULL IDENTITY PRIMARY KEY,
  qty  INT NOT NULL
);
```

```
CREATE INDEX idx_qty_empid ON dbo.SalesBM(qty, empid);
GO
CREATE TABLE dbo.SalesBMIdentity(empid INT, qty INT, rn INT IDENTITY);
GO
CREATE TABLE dbo.SalesBMCursor(empid INT, qty INT, rn INT);
GO

DECLARE
  @maxnumrows     AS INT,
  @steprows       AS INT,
  @curnumrows     AS INT,
  @dt             AS DATETIME;

SET @maxnumrows     = 100000;
SET @steprows       = 10000;
SET @curnumrows     = 10000;

WHILE @curnumrows <= @maxnumrows
BEGIN

  TRUNCATE TABLE dbo.SalesBM;
  INSERT INTO dbo.SalesBM(qty)
    SELECT CAST(1+999.9999999999*RAND(CHECKSUM(NEWID())) AS INT)
    FROM dbo.Nums
    WHERE n <= @curnumrows;

  -- 'Set-Based 2000'

  DBCC FREEPROCCACHE WITH NO_INFOMSGS;
  DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

  SET @dt = GETDATE();

  SELECT empid, qty,
    (SELECT COUNT(*)
     FROM dbo.SalesBM AS S2
     WHERE S2.qty < S1.qty
         OR (S2.qty = S1.qty AND S2.empid <= S1.empid)) AS rn
  FROM dbo.SalesBM AS S1
  ORDER BY qty, empid;
```

```
INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
  VALUES(1, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

-- 'IDENTITY'

TRUNCATE TABLE dbo.SalesBMIdentity;

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

SET @dt = GETDATE();

INSERT INTO dbo.SalesBMIdentity(empid, qty)
  SELECT empid, qty FROM dbo.SalesBM ORDER BY qty, empid;

SELECT empid, qty, rn FROM dbo.SalesBMIdentity;

INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
  VALUES(2, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

-- 'Cursor'

TRUNCATE TABLE dbo.SalesBMCursor;

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

SET @dt = GETDATE();

DECLARE @empid AS INT, @qty AS INT, @rn AS INT;

BEGIN TRAN

DECLARE rncursor CURSOR FAST_FORWARD FOR
  SELECT empid, qty FROM dbo.SalesBM ORDER BY qty, empid;
OPEN rncursor;

SET @rn = 0;

FETCH NEXT FROM rncursor INTO @empid, @qty;
WHILE @@fetch_status = 0
```

```
  BEGIN
    SET @rn = @rn + 1;
    INSERT INTO dbo.SalesBMCursor(empid, qty, rn)
      VALUES(@empid, @qty, @rn);
    FETCH NEXT FROM rncursor INTO @empid, @qty;
  END

  CLOSE rncursor;
  DEALLOCATE rncursor;

  COMMIT TRAN

  SELECT empid, qty, rn FROM dbo.SalesBMCursor;

  INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
    VALUES(3, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

  -- 'ROW_NUMBER 2005'

  DBCC FREEPROCCACHE WITH NO_INFOMSGS;
  DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

  SET @dt = GETDATE();

  SELECT empid, qty, ROW_NUMBER() OVER(ORDER BY qty, empid) AS rn
  FROM dbo.SalesBM;

  INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
    VALUES(4, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

  SET @curnumrows = @curnumrows + @steprows;

END
```

השאילתה הבאה מחזירה את תוצאות בוחן–הביצועים במבנה נוח לקריאה, המוצג בטבלה 4–29:

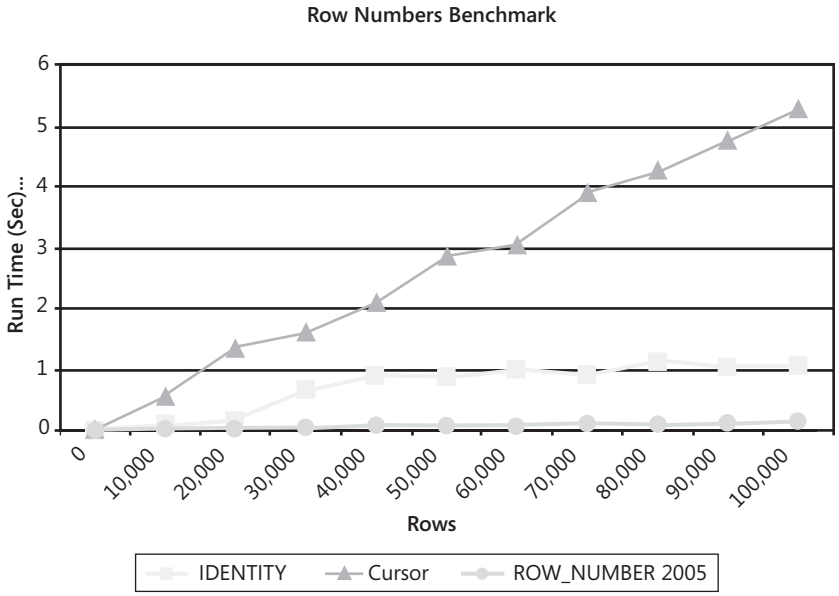
```
SELECT numrows,
  [Set-Based 2000], [IDENTITY], [Cursor], [ROW_NUMBER 2005]
FROM (SELECT technique, numrows, runtimems
      FROM dbo.RNBenchmark AS B
         JOIN dbo.RNTechniques AS T
            ON B.tid = T.tid) AS D
PIVOT(MAX(runtimems) FOR technique IN(
  [Set-Based 2000], [IDENTITY], [Cursor], [ROW_NUMBER 2005])) AS P
ORDER BY numrows;
```

טבלה 29–4: תוצאות בוחן–ביצועים

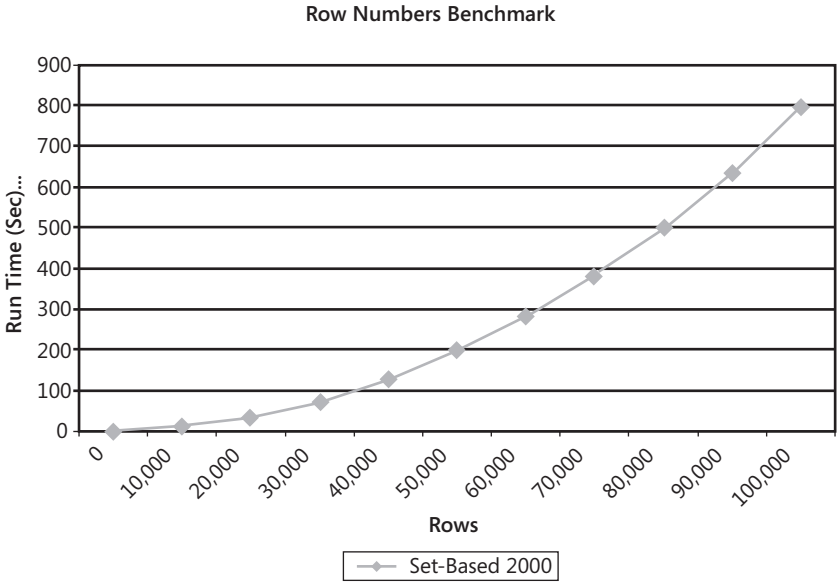
numrows	*Set-Based 2000*	*IDENTITY*	*Cursor*	*ROW_NUMBER 2005*
10000	11960	80	550	30
20000	33260	160	1343	40
30000	72646	660	1613	30
40000	127033	893	2110	60
50000	199740	870	2873	70
60000	283616	990	3043	63
70000	382130	900	3913	103
80000	499580	1123	4276	80
90000	634653	1040	4766	100
100000	796806	1060	5280	140

השאילתה משתמשת בשיטת סיבוב על ציר אותה אתאר בפרק 6, כך שאל תתאמץ אם אינך מבין אותה. לצורך הדיון שלנו, מה שחשוב הוא תוצאות בוחן–הביצועים. ניתן לראות מייד שהשיטה מבוססת–הסטים של טרום SQL Server 2005 איטית משמעותית מכל היתר, ומוקדם יותר הסברתי מדוע. תוכל גם להבחין שהפונקציה ROW_NUMBER מהירה משמעותית מכל היתר. התכוונתי להציג גרף המכיל את כל התוצאות, אך זמני הריצה של השיטות מבוססות–הסטים של טרום SQL Server 2005 היו כה איטיים עד כדי כך שהקווים של הפתרונות האחרים היו שטוחים. לפיכך החלטתי להציג שני גרפים נפרדים. תרשים 7–4 מציג את גרף זמני הריצה של השיטה המבוססת–IDENTITY, השיטה המבוססת–סמן והשיטה המבוססת–פונקציה ROW_MUNBER. תרשים 8–4 מציג את הגרף עבור השיטה מבוססת–הסטים טרום SQL Server 2005.

תרשים 7–4: בוחן–ביצועים למספרי שורה גרף I



תרשים 8–4: בוחן–ביצועים למספרי שורה גרף II



ניתן לראות בתרשים 7–4 שלכל שלוש השיטות יש גרף ביצועים ליניארי למדי, בעוד שתרשים 8–4 מראה גרף נאה של n^2.

המסקנה הברורה מאליה היא שב– SQL Server 2005 עליך להשתמש תמיד בפונקציות הדירוג החדשות. ב– SQL Server 2000, אם חשוב לך להשתמש בשיטה מבוססת–סטים התואמת לתקן, השתמש בשיטה זו רק כאשר גודל המחיצה קטן יחסית (נמדד בעשרות). אחרת, השתמש בשיטה מבוססת–IDENTITY, תוך שאתה ראשית יוצר את הטבלה ואז מכניס את הנתונים.

דפדוף (Paging)

כפי שהזכרתי קודם, למספרי שורה יישומים מעשיים רבים אותם אדגים לאורך הספר. כרגע ארצה להראות דוגמה אחת בה אני משתמש במספרי שורה כדי לבצע דפדוף – גישה לשורות של סט תוצאה במקטעים. דפדוף הוא צורך נפוץ ביישומים, המאפשר למשתמש לנווט דרך מקטעים או מנות של סט תוצאה. דפדוף עם מספרי שורה גם הוא שיטה שימושית. דוגמה זו גם תאפשר לי להדגים שיטות אופטימיזציה נוספות אותן מפעיל ה–optimizer כאשר הוא משתמש בפונקציה ROW_NUMBER. כמובן שב– SQL Server 2000 ניתן להשתמש בשיטות היעילות פחות לחישוב מספרי שורה לצורך השגת דפדוף.

דפדוף אד–הוק

דפדוף אד–הוק הוא בקשה לדף יחיד, כאשר הקלט הוא מספר הדף וגודל דף (מספר שורות בדף). כאשר המשתמש צריך דף יחיד מסוים ולא יבקש דפים נוספים, יש להשתמש בפתרון שונה מהפתרון בו תשתמש לבקשות דפים מרובים. ראשית חשוב להבין שאין כל דרך לגשת לדף n בלי לגשת פיסית לדפים 1 עד n–1. לאור עובדה זו, הקוד הבא מחזיר דף של שורות מטבלת Sales ממוינות לפי qty ו–empid, בהינתן גודל דף ומספר דף כקלט:

```
DECLARE @pagesize AS INT, @pagenum AS INT;
SET @pagesize = 5;
SET @pagenum = 2;

WITH SalesCTE AS
(
  SELECT ROW_NUMBER() OVER(ORDER BY qty, empid) AS rownum,
    empid, mgrid, qty
  FROM dbo.Sales
)
SELECT rownum, empid, mgrid, qty
FROM SalesCTE
WHERE rownum > @pagesize * (@pagenum-1)
  AND rownum <= @pagesize * @pagenum
ORDER BY rownum;
```

קוד זה מייצר את הפלט המוצג בטבלה 30–4.

טבלה 30–4: דף שני של Sales ממוין לפי qty, empid עם גודל דף של 5 שורות

rownum	*empid*	*mgrid*	*qty*
6	D	Y	200
7	K	Y	200
8	E	Z	250
9	I	X	250
10	A	Z	300

ה–CTE שנקרא SalesCTE מקצה מספרי שורה לשורות המכירה בהתבסס על המיון של qty ו–empid. השאילתה החיצונית מסננת רק את השורות של דף היעד תוך שימוש בנוסחה המתבססת על מספר דף הקלט וגודלו.

ייתכן שתחשוש כי השאילתה מחשבת מספרי שורה לכל השורות ואז מסננת רק את השורות של הדף המבוקש. עלול להיראות שהדבר דורש סריקה מלאה של הטבלה. עם טבלאות גדולות מאוד יהיו לכך כמובן השלכות חמורות על הביצועים. אף על פי כן, בטרם תתחיל לחשוש, בחן את תוכנית העבודה עבור שאילתה זו, המוצגת בתרשים 9–4.

תרשים 9–4: תוכנית עבודה לפתרון לדפדוף אד–הוק

התרשים מראה רק את החלק השמאלי של התוכנית המתחיל ב– Sequence Project המקצה את מספרי השורה. אם תתבונן במאפיינים של האופרטור TOP, תראה שהתוכנית סורקת רק את 10 השורות הראשונות בטבלה. מכיוון שהקוד מבקש את הדף השני של 5 שורות, רק שני הדפים הראשונים נסרקים. אז האופרטור Filter מסנן רק את הדף השני (שורות 6 עד 10).

דרך נוספת להראות שלא כל הטבלה נסרקת היא למלא את הטבלה במספר גדול של שורות ולהריץ את השאילתה כאשר האפשרות SET STATISTICS IO מופעלת. תוכל להבחין לפי מספר הקריאות המדווחות שכאשר אתה מבקש את דף n, ללא קשר לגודל הטבלה, רק n הדפים הראשונים של השורות נסרקים.

לפתרון יש ביצועים טובים אפילו עבור בקשות מרובות לדפים שלרוב "נעות קדימה" – כלומר, מבוקש דף 1, לאחר מכן דף 2, לאחר מכן דף 3 וכך הלאה. כאשר מבוקש דף השורות הראשון, הנתונים/האינדקס הרלוונטיים נסרקים פיסית ונטענים לזיכרון (אם הם עדיין אינם שם). כאשר מבוקש דף השורות השני, דפי הנתונים לבקשה הראשונה נמצאים כבר

בזיכרון, ורק דפי הנתונים לדף השורות השני צריכים להיסרק פיסית. דבר זה דורש בעיקר קריאות לוגיות (קריאה מהזיכרון), וקריאות פיסיות נדרשות רק עבור הדף המבוקש. קריאות לוגיות הן מהירות הרבה יותר מאשר קריאות פיסיות, אך זכור שגם להן עלות מצטברת.

גישה לדפים מרובים

קיים פתרון נוסף לדפדוף שלרוב יהיו לו באופן כללי ביצועים טובים יותר מאשר לפתרון הקודם, כאשר קיימות בקשות למספר דפים שאינם מתקדמים קדימה, כאשר סט התוצאה אינו גדול במיוחד. ראשית ממש (materialize) את כל העמודים בטבלה בצירוף מספרי השורה וצור אינדקס clustered– על טור מספר השורה:

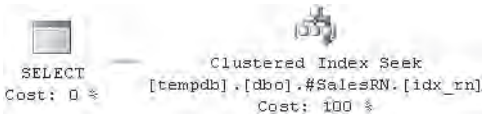
```
SELECT ROW_NUMBER() OVER(ORDER BY qty, empid) AS rownum,
  empid, mgrid, qty
INTO #SalesRN
FROM dbo.Sales;

CREATE UNIQUE CLUSTERED INDEX idx_rn ON #SalesRN(rownum);
```

כעת תוכל למלא כל בקשה לדף עם שאילתה כלהלן:

```
DECLARE @pagesize AS INT, @pagenum AS INT;
SET @pagesize = 5;
SET @pagenum = 2;

SELECT rownum, empid, mgrid, qty
FROM #SalesRN
WHERE rownum BETWEEN @pagesize * (@pagenum-1) + 1
                 AND @pagesize *  @pagenum
ORDER BY rownum;
```

תוכנית העבודה לשאילתה זו מוצגת בתרשים 10–4 (מקוצרת על ידי הסרת האופרטורים לחישוב גבולות עד לאופרטור Merge Interval, כדי להתמקד בגישה לנתונים בלבד).

תרשים 10–4: תוכנית עבודה לפתרון לדפים מרובים



זוהי תוכנית יעילה ביותר המבצעת חיפוש בתוך האינדקס להגיע לשורה שבגבול התחתון (שורה 6 במקרה זה), ולאחר מכן סריקה חלקית (לא נראית בתוכנית), עד שמגיעה לשורה שבגבול העליון (שורה 10). רק השורות של דף התוצאות המבוקש נסרקות בתוך האינדקס.

אם עיצוב היישום שלך הוא כזה שמתנתק לאחר כל בקשה, כמובן שהטבלה הזמנית תיעלם ברגע שה–session היוצר מתנתק. במקרה כזה, ייתכן שתרצה ליצור טבלה קבועה שהיא "זמנית" לוגית. ניתן לבצע זאת על ידי שינוי שם הטבלה some_name<some_identifier> – למשל, T<guid> (Global Unique Identifier).

יהיה עליך גם לפתח תהליך ניקוי, הנפטר מהטבלאות אשר היישום לא הספיק להסיר במפורש במקרים בהם הוא הסתיים בצורה לא מסודרת.

במקרים בהם עליך לתמוך בסטים גדולים של תוצאות או ברמה גבוהה של עבודה בריבוי משתמשים, יהיו לך סוגיות של גדילת הפעילות (scalability) הקשורות למשאבים של tempdb. באפשרותך לפתח פתרון במחיצות הממשמש רק מספר מסוים של דפים ולא את כולם – למשל, 1000 שורות בכל פעם. בדרך כלל, משתמשים אינם מבקשים יותר מאשר את מספר הדפים הראשונים בכל מקרה. אם משתמש אכן מבקש דפים מעבר לקבוצה הראשונה, תוכל לממש את הקבוצה הבאה (כלומר, את 1000 השורות הבאות).

כאשר לא חשוב לך לממש את סט התוצאה בטבלה זמנית לצורך גישה לדפים מרובים, ייתכן שתשקול להשתמש במשתנה טבלה בו אתה מממש רק את הקבוצה הראשונה של הדפים (למשל, 1000 שורות). משתני טבלה אינם מערבים קומפילציות חוזרות, והם סובלים פחות מסוגיות של תיעוד ונעילות. ה–optimizer אינו אוסף סטטיסטיקות עבור משתני טבלה, כך שעליך להיות זהיר מאוד וברור בבחירת המקרים בהם תשתמש בהם. אך כאשר כל שעליך לעשות הוא לאחסן סט תוצאה קטן ולסרוק את כולו בכל מקרה, הדבר סביר.

כאשר תסיים להשתמש בטבלה זו, תוכל להסיר אותה:

```
DROP TABLE #SalesRN;
```

RANK ו–DENSE RANK

RANK ו– DENSE RANK הם חישובים דומים למספרי שורה. אך שלא כמו מספרי שורה, להם מגוון רחב של פתרונות מעשיים, RANK ו– DENSE RANK משמשים באופן טיפוסי ליישומים של דירוג וניקוד.

פונקציות RANK ו–DENSE_RANK ב– SQL Server 2005

SQL Server 2005 מספק לך פונקציות מובנות של RANK ו– DENSE_RANK הדומות לפונקציה ROW_NUMBER. ההבדל בין פונקציות אלו לבין ROW_NUMBER הוא, כפי שהסברתי קודם, בכך ש– ROW_NUMBER אינה דטרמיניסטית כאשר רשימת ORDER BY אינה ייחודית. RANK ו– DENSE_RANK הן תמיד דטרמיניסטיות – כלומר, אותם ערכי דירוג מוקצים לשורות בעלות ערך מיון זהה. ההבדל בין RANK ל–DENSE_RANK

הוא שב–RANK עשויים להיות פערים בערכי הדירוג, אך היא מאפשרת לך לדעת לכמה שורות יש ערכי מיון נמוכים יותר. בערכי DENSE_RANK אין פערים.

לשם הדוגמה, השאילתה הבאה (המייצרת את הפלט המוצג בטבלה 31–4) מחזירה ערכים הן של RANK והן של DENSE RANK עבור שורות המכירה בהתבסס על מיון לפי כמות:

```
SELECT empid, qty,
  RANK() OVER(ORDER BY qty) AS rnk,
  DENSE_RANK() OVER(ORDER BY qty) AS drnk
FROM dbo.Sales
ORDER BY qty;
```

טבלה 31–4: Rank ו– Dense Rank

empid	*qty*	*rnk*	*drnk*
B	100	1	1
G	100	1	1
J	100	1	1
H	150	4	2
C	200	5	3
D	200	5	3
K	200	5	3
E	250	8	4
I	250	8	4
A	300	10	5
F	300	10	5

הרי לך חידה קצרה: מה ההבדל בין התוצאות של ROW_NUMBER, RANK ו–DENSE_RANK בהינתן רשימה ייחודית של ORDER BY?

כדי לענות, הרץ את הקוד הבא:

```
SELECT REVERSE('!gnihton yletulosbA');
```

פתרונות מבוססי–סטים טרום SQL Server 2005

חישובים של RANK ו– DENSE RANK המשתמשים בשיטה מבוססת–סטים טרום SQL Server 2005 דומים מאוד לחישובים של מספרי שורה. כדי לחשב RANK, השתמש בתת–שאילתה הסופרת את מספר השורות בעלות ערך המיון הנמוך יותר, ומוסיפה אחד. כדי לחשב DENSE RANK, השתמש בתת–שאילתה הסופרת את המספר הייחודי של ערכי מיון נמוכים ומוסיפה אחד.

```
SELECT empid, qty,
  (SELECT COUNT(*) FROM dbo.Sales AS S2
   WHERE S2.qty < S1.qty) + 1 AS rnk,
  (SELECT COUNT(DISTINCT qty) FROM dbo.Sales AS S2
   WHERE S2.qty < S1.qty) + 1 AS drnk
FROM dbo.Sales AS S1
ORDER BY qty;
```

כמובן, ביכולתך להוסיף קישור להחזרת חישובים במחיצות בדיוק כפי שעשית עם מספרי שורה.

NTILE

הפונקציה NTILE מחלקת שורות למספר מסוים של אריחים (או קבוצות). האריחים ממוספרים מ–1 והלאה. לכל שורה מוקצה מספר האריח אליו היא שייכת. NTILE מבוססת על חישוב מספרי שורה ומיון מבוקש. הפונקציה יכולה, אופציונלית, להיות מחושבת במחיצות, וכך החלוקה לאריחים תתבצע בנפרד בכל מחיצה. בהתבסס על מספר השורות בטבלה (או במחיצה), מספר האריחים המבוקשים, ומספר השורה, ניתן לקבוע את מספר האריח לכל שורה. למשל, לטבלה בת 10 שורות, הערך של NTILE(2) OVER (ORDER BY c) יהיה 1 עבור 5 השורות הראשונות במיון טור c, ו–2 עבור שורות 6 עד 10.

באופן טיפוסי, חישובי NTILE משמשים לצרכי ניתוח כגון חישוב אחוזונים או סידור פריטים בקבוצות.

למשימה של "ריצוף" (tiling) יש יותר מפתרון אחד, ו– SQL Server 2005 מיישם פתרון ספציפי הנקרא ANSI NTILE. ראשית אתאר את היישום של הפונקציה NTILE, ואז אעבור לפתרונות אחרים.

הפונקציה NTILE ב– SQL Server 2005

חישוב NTILE ב– SQL Server 2005 הוא פשוט כמו החישוב של כל ערכי הדירוג האחרים – במקרה זה, בעזרת שימוש בפונקציה NTILE. ההבדל היחיד הוא ש–NTILE מקבלת קלט, מספר האריחים, בעוד שפונקציות הדירוג האחרות אינן מקבלות קלט.

מכיוון שחישובי NTILE מבוססים על מספרי שורה, NTILE נתקלת בדיוק באותן סוגיות דטרמיניזם אותן תיארתי בנושא מספרי שורה.

לשם הדוגמה, השאילתה הבאה מחשבת ערכי NTILE לשורות מהטבלה Sales, ומייצרת שלושה אריחים, בהתבסס על מיון של qty ו–empid:

```
SELECT empid, qty,
  NTILE(3) OVER(ORDER BY qty, empid) AS tile
FROM dbo.Sales
ORDER BY qty, empid;
```

טבלה 32–4: פלט של שאילתה NTILE

empid	*qty*	*Tile*
B	100	1
G	100	1
J	100	1
H	150	1
C	200	2
D	200	2
K	200	2
E	250	2
I	250	3
A	300	3
F	300	3

שים לב שכאשר מספר האריחים (num_tiles) לא מחלק את מספר השורות בטבלה (cnt) בצורה שווה, r האריחים הראשונים (כאשר r הוא cnt%num_tiles) מקבלים שורה אחת יותר מהאחרים. במילים אחרות, השארית מוקצית לאריחים הראשונים קודם. בדוגמה שלנו, בטבלה 11 שורות, ונדרשו שלושה אריחים. הגודל הבסיסי של אריח הוא 3 = 3 / 11 (חלוקת שלמים). השארית היא 2 = 3 % 11. האופרטור % (modulo) מספק את השלם שהוא שארית לאחר חלוקה של השלם הראשון בשני. כך ששני האריחים הראשונים מקבלים שורה נוספת מעבר לגודל הבסיס של אריח ומסיימים עם ארבע שורות.

כדוגמה משמעותית יותר, נניח שעליך לפצל את שורות המכירה לשלוש קטגוריות בהתבסס על כמויות: נמוכה, בינונית וגבוהה. ברצונך שבכל קטגוריה תהיה בערך כמות זהה של שורות. תוכל לחשב ערכי NTILE(3) בהתבסס על מיון qty (ושימוש ב–empid כשובר–שוויון רק כדי להבטיח תוצאות דטרמיניסטיות הניתנות להפקה חוזרת) ולהשתמש

בביטוי CASE להמרת מספרי האריחים לתיאור משמעותי יותר:

```
SELECT empid, qty,
  CASE NTILE(3) OVER(ORDER BY qty, empid)
    WHEN 1 THEN 'low'
    WHEN 2 THEN 'medium'
    WHEN 3 THEN 'high'
  END AS lvl
FROM dbo.Sales
ORDER BY qty, empid;
```

הפלט של שאילתה זו מוצג בטבלה 33–4.

טבלה 33–4: אריחים תיאוריים

empid	*qty*	*lvl*
B	100	low
G	100	low
J	100	low
H	150	low
C	200	medium
D	200	medium
K	200	medium
E	250	medium
I	250	high
A	300	high
F	300	high

לחישוב הטווח של כמויות השייכות לכל קטגוריה כפי שמוצג בטבלה 34–4, פשוט קבץ את הנתונים לפי מספר האריח, כך שיחזרו ערכי המיון המינימלי והמקסימלי לכל קבוצה:

```
WITH Tiles AS
(
  SELECT empid, qty,
    NTILE(3) OVER(ORDER BY qty, empid) AS tile
  FROM dbo.Sales
)
SELECT tile, MIN(qty) AS lb, MAX(qty) AS hb
FROM Tiles
GROUP BY tile
ORDER BY tile;
```

טבלה 34–4: טווחים של כמויות השייכות לכל קטגוריה

tile	*lb*	*hb*
1	100	150
2	200	250
3	250	300

פתרונות מבוססי–סטים אחרים ל–NTILE

חישוב ANSI NTILE קודם ל– SQL Server 2005 תוך שימוש בשיטות מבוססות–סטים הוא קשה יותר וכמובן שהרבה יותר יקר. חישוב סוגים אחרים של אריחים אינו מובן מאליו אפילו ב– SQL Server 2005.

הנוסחה בה אתה משתמש לחישוב NTILE תלויה במה שברצונך לעשות בדיוק עם השארית במקרה שמספר השורות בטבלה לא מתחלק בצורה שווה במספר האריחים. ייתכן שתרצה להשתמש בגישה של הפונקציה ANSI NTILE, שאומרת, "פשוט הקצה את השארית לאריחים הראשונים, שורה לכל אריח עד שכולן מתכלות". שיטה אחרת, שסטטיסטית ייתכן כי היא נכונה יותר, היא לחלק את השארית בין האריחים בפיזור שוויוני יותר במקום לשים אותם באריחים הראשונים בלבד.

אתחיל בשיטה השנייה, חישוב ערכי NTILE בפיזור שוויוני, מכיוון שהיא פשוטה יותר. נדרשים לך שני קלטים לחישוב מספר האריח לשורה: מספר השורה וגודל האריח. כיצד לחשב מספרי שורה אתה כבר יודע. לחישוב גודל אריח אתה מחלק את מספר השורות בטבלה במספר המבוקש של אריחים. הנוסחה שמחשבת את מספר אריח היעד היא:

```
(row_number - 1) / tile_size + 1
```

הטריק שיאפשר לך לחלק את השארית בצורה שווה הוא להשתמש בחישוב עשרוני לחשב את הערך tile_size, במקום בחישוב שלמים. כלומר, במקום להשתמש בחישוב שלמים עבור גודל האריח (num_rows/num_tiles), שיקטע את השבר, השתמש ב– 1.*num_rows/num_tiles, אשר יחזיר תוצאה עשרונית נכונה יותר. לבסוף, כדי להיפטר מהשבר במספר האריח, המר את התוצאה חזרה לערך שלם.

להלן השאילתה השלמה המייצרת מספרי אריחים תוך שימוש בגישת הפיזור–השוויוני ומייצרת את הפלט המוצג בטבלה 35–4:

```
DECLARE @numtiles AS INT;
SET @numtiles = 3;

SELECT empid, qty,
  CAST((rn - 1) / tilesize + 1 AS INT) AS tile
FROM (SELECT empid, qty, rn,
        1.*numrows/@numtiles AS tilesize
      FROM (SELECT empid, qty,
              (SELECT COUNT(*) FROM dbo.Sales AS S2
               WHERE S2.qty < S1.qty
                 OR S2.qty = S1.qty
                   AND S2.empid <= S1.empid) AS rn,
              (SELECT COUNT(*) FROM dbo.Sales) AS numrows
            FROM dbo.Sales AS S1) AS D1) AS D2
ORDER BY qty, empid;
```

טבלה 35–4: NTILE, פיזור–שוויוני של שארית, **3** אריחים

empid	*qty*	*tile*
B	100	1
G	100	1
J	100	1
H	150	1
C	200	2
D	200	2
K	200	2
E	250	2
I	250	3
A	300	3
F	300	3

עם שלושה אריחים, אינך יכול לראות את הפיזור השוויוני של שורות השארית. אם תריץ קוד זה תוך שימוש בתשעה אריחים כקלט, תקבל את הפלט המוצג בטבלה 36–4 שם הפיזור השוויוני ברור יותר.

טבלה 36–4: NTILE, פיזור שוויוני של שארית, 9 אריחים

empid	*qty*	*tile*
B	100	1
G	100	1
J	100	2
H	150	3
C	200	4
D	200	5
K	200	5
E	250	6
I	250	7
A	300	8
F	300	9

תוכל לראות בתוצאה שהאריח הראשון מכיל שתי שורות, שלושת האריחים הבאים מכילים שורה אחת כל אחד, האריח הבא מכיל שתי שורות וארבעת האריחים האחרונים מכילים שורה אחת כל אחד. תוכל להתנסות עם הקלט של מספר האריחים להשגת תמונה ברורה יותר של האלגוריתם לפיזור–שוויוני.

כדי להשיג תוצאה זהה לזו של הפונקציה ANSI NTILE, כאשר השארית מחולקת לאריחים הממוספרים ראשונים, עליך להשתמש בנוסחה שונה. ראשית, החישובים מערבים שלמים בלבד. הקלטים להם אתה נדרש במקרה זה כוללים את מספר השורה, גודל האריח והשארית (number of rows in the table % number of requested tiles). קלטים אלו משמשים לחישוב NTILE בפיזור לא–שוויוני.

הנוסחה למספר אריח היעד היא כלהלן:

```
if row_number <= (tilesize + 1) * remainder then
  tile_number = (row_number - 1) / (tile_size + 1) + 1
else
  tile_number = (row_number - remainder - 1) / tile_size + 1
```

בתרגום ל–T-SQL, השאילתה (המייצרת את הפלט המוצג בטבלה 37–4) נראית כך:

```
DECLARE @numtiles AS INT;
SET @numtiles = 9;

SELECT empid, qty,
  CASE
    WHEN rn <= (tilesize+1) * remainder
      THEN (rn-1) / (tilesize+1) + 1
    ELSE (rn - remainder - 1) / tilesize + 1
  END AS tile
FROM (SELECT empid, qty, rn,
        numrows/@numtiles AS tilesize,
        numrows%@numtiles AS remainder
      FROM (SELECT empid, qty,
              (SELECT COUNT(*) FROM dbo.Sales AS S2
               WHERE S2.qty < S1.qty
                 OR S2.qty = S1.qty
                   AND S2.empid <= S1.empid) AS rn,
              (SELECT COUNT(*) FROM dbo.Sales) AS numrows
            FROM dbo.Sales AS S1) AS D1) AS D2
ORDER BY qty, empid;
```

טבלה 37–4: NTILE, שארית נוספת לקבוצות הראשונות

empid	*qty*	*tile*
B	100	1
G	100	1
J	100	2
H	150	2
C	200	3
D	200	4
K	200	5
E	250	6
I	250	7
A	300	8
F	300	9

הפלט זהה לזה שתקבל על ידי שימוש בפונקציה NTILE של SQL Server 2005; האריחים הראשונים מקבלים שורה נוספת עד שהשארית מתכלה.

טבלת עזר של מספרים

טבלת עזר של מספרים היא כלי חזק מאוד בו אני משתמש לעיתים קרובות בפתרונות שלי, לכן החלטתי להקדיש לה סעיף מפרק זה. בסעיף זה פשוט אתאר את המושג ואת השיטות המשמשות ליצירת טבלה כזו. אתייחס לטבלה זו לאורך כל הספר ואדגים רבים מהיישומים שלה.

טבלת עזר של מספרים (נקרא לה Nums) היא פשוט טבלה המכילה את השלמים שבין 1 ל–N לערך כלשהו (לרוב גדול) של N. אני ממליץ לך ליצור טבלת Nums קבועה ולהכניס בה ערכים רבים ככל שתצטרך לפתרונות שלך.

הקוד בקטע–קוד 8–4 מדגים כיצד לייצר טבלה כזו בת 1,000,000 שורות. כמובן שייתכן שתרצה מספר שונה של שורות, בהתאם לצרכים שלך.

קטע–קוד 8–4: יצירה ומילוי של טבלת עזר של מספרים

```
SET NOCOUNT ON;
USE AdventureWorks;
GO
IF OBJECT_ID('dbo.Nums') IS NOT NULL
  DROP TABLE dbo.Nums;
GO
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);
DECLARE @max AS INT, @rc AS INT;
SET @max = 1000000;
SET @rc = 1;

INSERT INTO Nums VALUES(1);
WHILE @rc * 2 <= @max
BEGIN
  INSERT INTO dbo.Nums SELECT n + @rc FROM dbo.Nums;
  SET @rc = @rc * 2;
END

INSERT INTO dbo.Nums
  SELECT n + @rc FROM dbo.Nums WHERE n + @rc <= @max;
```

טיפ: מכיוון שיש שימושים מעשיים כה רבים לטבלת Nums, סביר להניח שתצטרך לגשת אליה ממסדי נתונים שונים. כדי להימנע מהצורך לגשת אליה בעזרת שימוש בשם המלא AdventureWorks.dbo.Nums, תוכל ליצור שם נרדף במסד הנתונים model המצביע על Nums ב– AdventureWorks בצורה הבאה:

```
USE model;
CREATE SYNONYM dbo.Nums FOR AdventureWorks.dbo.Nums;
```

יצירת השם הנרדף ב–model יגרום לו להיות זמין בכל מסדי הנתונים החדשים שייוצרו מרגע זה ואילך, כולל tempdb לאחר ש– SQL Server מאותחל. עבור מסדי נתונים קיימים, תצטרך פשוט להריץ פעם אחת במפורש את הפקודה CREATE SYNONYM.

מעשית, לא ממש משנה כיצד אתה מכניס נתונים לטבלה Nums מכיוון שאתה מריץ את התהליך הזה פעם אחת בלבד. אף על פי כן, השתמשתי בתהליך אופטימלי המכניס נתונים לטבלה בצורה מהירה ביותר. התהליך מדגים את השיטה של יצירת Nums עם לולאת INSERT מכפילה.

הקוד שומר את מספר השורות שהוכנסו כבר לטבלה במשתנה שנקרא @rc. הוא ראשית מכניס ל–Nums את השורה שבה n=1. אז הוא נכנס ללולאה כל עוד @max => 2 * @rc (@max הוא מספר השורות הרצוי). בכל חזרה, התהליך מכניס ל–Nums את התוצאה של שאילתה הבוחרת את כל השורות מ–Nums לאחר הוספת @rc לכל ערך n. שיטה זו מכפילה את מספר השורות ב–Nums בכל חזרה – כלומר, ראשית מוכנס {1}, לאחר מכן {2}, לאחר מכן {3, 4}, לאחר מכן {5, 6, 7, 8}, לאחר מכן {9, 10, 11, 12, 13, 14, 15, 16} וכך הלאה.

ברגע שבטבלה יש שורות בכמות שהיא יותר ממחצית מספר היעד של שורות, הלולאה עוצרת. משפט INSERT נוסף לאחר הלולאה מכניס את השורות שנותרו תוך שימוש באותו משפט INSERT כמו זה שבתוך הלולאה, אך הפעם עם מסנן כדי להבטיח שיתווספו רק ערכים הקטנים או שווים ל–@max.

הסיבה המרכזית בגללה תהליך זה רץ מהר היא שהוא דורש את הכמות המינימלית של כתיבות ל– transaction log בהשוואה לפתרונות אחרים. דבר זה מושג על ידי שימוש במינימום של משפטי INSERT (מספר משפטי ה–INSERT הוא CEILING(LOG2(@MAX)) + 1). קוד זה הכניס לטבלת Nums 1,000,000 שורות בתוך 6 שניות על המחשב שלי. כתרגיל, תוכל לנסות להכניס נתונים לטבלת Nums תוך שימוש בלולאה רגילה של הכנסות יחידניות ולראות כמה זמן זה לוקח.

בכל פעם שאתה צריך את @n המספרים הראשונים מתוך Nums, פשוט הפעל עליה שאילתה, והגדר כמסנן WHERE n <= @n. אינדקס על הטור n יבטיח שהשאילתה תסרוק רק את השורות המבוקשות ולא אחרות.

אם אינך מורשה להוסיף טבלאות קבועות למסד הנתונים, תוכל ליצור UDF טבלאי עם פרמטר למספר השורות הנדרש. אתה משתמש בלוגיקה זהה לזו בה השתמשת קודם ליצירת מספר הערכים הנדרש.

ב– SQL Server 2005 באפשרותך להשתמש ב–CTE ובפונקציה ROW_NUMBER החדשים, ליצירת פתרונות יעילים ביותר המייצרים טבלת מספרים בזמן ריצה.

אתחיל בפתרון נאיבי איטי יחסית (בערך 20 שניות, ללא יצירת הפלט). הפתרון הבא משתמש ב–CTE רקורסיבי פשוט, כאשר איבר העוגן מייצר שורה עם n=1, והאיבר הרקורסיבי מוסיף שורה בכל חזרה עם n = prev n + 1:

```
DECLARE @n AS BIGINT;
SET @n = 1000000;

WITH Nums AS
(
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM Nums WHERE n < @n
)
SELECT n FROM Nums
OPTION(MAXRECURSION 0);
```

שים לב: אם אתה מריץ את הקוד כדי לנסות אותו, זכור להדליק ב–SSMS את האפשרות של Discard Results After Execution; אחרת, תקבל פלט עם מיליון שורות.

באפשרותך לשפר את הפתרון בצורה משמעותית על ידי שימוש ב–CTE (נקרא לו Base) המייצר שורות בכמות של השורש הריבועי של מספר היעד של שורות. השתמש בהצלבה של שני מופעים של Base לקבלת מספר היעד של שורות, ולבסוף, יצר מספרי שורה עבור התוצאה שישמשו כרצף של המספרים.

להלן הקוד המיישם גישה זו:

```
DECLARE @n AS BIGINT;
SET @n = 1000000;

WITH Base AS
(
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM Base WHERE n < CEILING(SQRT(@n))
),
```

```
Expand AS
(
  SELECT 1 AS c
  FROM Base AS B1, Base AS B2
),
Nums AS
(
  SELECT ROW_NUMBER() OVER(ORDER BY c) AS n
  FROM Expand
)
SELECT n FROM Nums WHERE n <= @n
OPTION(MAXRECURSION 0);
```

פתרון זה רץ 0.8 שניות בלבד (ללא יצירת הפלט).

כעת אתאר את הגישה השלישית ליצירת Nums. אתה מתחיל עם CTE שלו שתי שורות בלבד, ומכפיל את מספר השורות עם כל CTE נוסף על ידי הצלבת שני מופעים של ה–CTE הקודם. עם n רמות של CTE (מבוססי–0), אתה מגיע ל–2^{2n}. למשל, עם 5 רמות, אתה מקבל 4,294,967,296 שורות.

CTE אחר מייצר מספרי שורה, ולבסוף השאילתה החיצונית מסננת את מספר הערכים הרצוי (כאשר row number column <= input). זכור שכאשר אתה מסנן ערך row number <= some value, SQL Server לא טורח לייצר מספרי שורה מעבר לנקודה זו. כך שאין עליך לדאוג לגבי ביצועים. זה לא מקרה בו הקוד שלך באמת ייצר למעלה מארבעה מיליארד שורות בכל פעם ואז יסנן.

להלן הקוד שמיישם גישה זו:

```
DECLARE @n AS BIGINT;
SET @n = 1000000;

WITH
L0   AS(SELECT 1 AS c UNION ALL SELECT 1),
L1   AS(SELECT 1 AS c FROM L0 AS A, L0 AS B),
L2   AS(SELECT 1 AS c FROM L1 AS A, L1 AS B),
L3   AS(SELECT 1 AS c FROM L2 AS A, L2 AS B),
L4   AS(SELECT 1 AS c FROM L3 AS A, L3 AS B),
L5   AS(SELECT 1 AS c FROM L4 AS A, L4 AS B),
Nums AS(SELECT ROW_NUMBER() OVER(ORDER BY c) AS n FROM L5)
SELECT n FROM Nums WHERE n <= @n;
```

הקוד רץ בערך 0.9 שניות כדי לייצר רצף של 1,000,000 מספרים.

כפי שציינתי קודם, תוכל לארוז את הלוגיקה ב–UDF. הערך של פתרון זה שהוא לא יתקרב לגבול של MAXRECURSION, וגבול זה אינו יכול להיות מצוין בתוך הגדרת UDF. הקוד בקטע–קוד 9–4 עוטף את הלוגיקה של הפתרון האחרון בתוך UDF.

קטע–קוד 9–4: UDF המחזיר טבלת עזר של מספרים

```
CREATE FUNCTION dbo.fn_nums(@n AS BIGINT) RETURNS TABLE
AS
RETURN
  WITH
  L0   AS(SELECT 1 AS c UNION ALL SELECT 1),
  L1   AS(SELECT 1 AS c FROM L0 AS A, L0 AS B),
  L2   AS(SELECT 1 AS c FROM L1 AS A, L1 AS B),
  L3   AS(SELECT 1 AS c FROM L2 AS A, L2 AS B),
  L4   AS(SELECT 1 AS c FROM L3 AS A, L3 AS B),
  L5   AS(SELECT 1 AS c FROM L4 AS A, L4 AS B),
  Nums AS(SELECT ROW_NUMBER() OVER(ORDER BY c) AS n FROM L5)
  SELECT n FROM Nums WHERE n <= @n;
GO
```

כדי לבחון את הפונקציה, הרץ את הקוד הבא, המחזיר טבלת עזר בת 10 מספרים:

```
SELECT * FROM dbo.fn_nums(10) AS F;
```

טווחים קיימים וחסרים (Islands ו–Gaps)

טווחים קיימים וחסרים ידועים גם כאיים – Islands ופערים – Gaps. כדי להפעיל את הידע שלך בנושא תת–שאילתות, ביטויי טבלה וחישובי דירוג, אציג זוג בעיות להן יישומים רבים בסביבות תפעוליות. אציג צורה כללית של הבעיה כדי שתוכל להתמקד בשיטות ולא בנתונים.

צור ומלא את הטבלה T1 על ידי הרצת הקוד בקטע–קוד 10–4.

קטע קוד 4–10: יצירה ומילוי של טבלה T1

```
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1
GO
CREATE TABLE dbo.T1(col1 INT NOT NULL PRIMARY KEY);
INSERT INTO dbo.T1(col1) VALUES(1);
INSERT INTO dbo.T1(col1) VALUES(2);
INSERT INTO dbo.T1(col1) VALUES(3);
INSERT INTO dbo.T1(col1) VALUES(100);
INSERT INTO dbo.T1(col1) VALUES(101);
INSERT INTO dbo.T1(col1) VALUES(103);
INSERT INTO dbo.T1(col1) VALUES(104);
INSERT INTO dbo.T1(col1) VALUES(105);
INSERT INTO dbo.T1(col1) VALUES(106);
```

לפניך שתי משימות. המשימה הראשונה היא להחזיר את טווחי המפתחות החסרים בנתונים, תוך יצירת הפלט המוצג בטבלה 4–38.

טבלה 4–38: טווחים חסרים

start_range	*end_range*
4	99
102	102

המשימה השנייה היא להחזיר טווחים של מפתחות רציפים בנתונים, תוך יצירת הפלט המופיע בטבלה 4–39.

טבלה 4–39: טווחים קיימים

start_range	*end_range*
1	3
100	101
103	106

בעיות אלו באות לידי ביטוי במערכות תפעוליות בצורות רבות – למשל, דוחות זמינות או אי–זמינות. במקרים מסוימים, הערכים מופיעים כשלמים כמו בדוגמה שלנו. במקרים אחרים, הם מופיעים כערכי datetime. השיטות הנדרשות לפתרון הבעיה עם שלמים ישימות לערכי datetime עם שינויים קלים בלבד.

טווחים חסרים (ידועים גם כפערים – Gaps)

קיימות מספר גישות לפתרון הבעיה של טווחים חסרים (פערים), אך ראשית חשוב לזהות את השלבים בפתרון בטרם מתחילים לקודד.

גישה אחת ניתן לתאר על ידי השלבים הבאים:

- מצא את הנקודות לפני הפערים, והוסף אחד לכל אחת מהן.
- לכל נקודת התחלה של פער, מצא את הערך הבא הקיים בטבלה והחסר אחד.

לאחר שזיהינו את ההיבטים הלוגיים, ניתן להתחיל לקודד. בצעדים הלוגיים הקודמים תמצא שהפרק כיסה את כל שיטות היסוד המוזכרות – בפרט, מציאת "נקודות לפני פערים" ומציאת הערך הקיים "הבא".

השאילתה הבאה מחזירה את הנקודות לפני הפערים המופיעות בטבלה 40–4:

```
SELECT col1
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.col1 = A.col1 + 1);
```

טבלה 40–4: נקודות לפני פער

col1
3
101
106

זכור שנקודה לפני פער היא ערך שהבא אחריו אינו קיים.

שים לב בפלט שהשורה האחרונה אינה מעניינת אותנו מכיוון שהיא לפני אינסוף. השאילתה הבאה מחזירה את נקודות ההתחלה של פערים, והפלט שלה מוצג בטבלה 41–4. היא השיגה זאת על ידי הוספת אחד לנקודות לפני הפערים להשגת הערך הראשון בפער, תוך כדי שהיא מסננת החוצה את הנקודה לפני אינסוף.

```
SELECT col1 + 1 AS start_range
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.col1 = A.col1 + 1)
  AND col1 < (SELECT MAX(col1) FROM dbo.T1);
```

טבלה 41–4: נקודות התחלה של פערים

start_range
4
102

לבסוף, לכל נקודת התחלה של פער, אתה משתמש בתת–שאילתה להחזרת הערך הבא בטבלה פחות 1 – במילים אחרות, סוף הפער:

```
SELECT col1 + 1 AS start_range,
  (SELECT MIN(col1) FROM dbo.T1 AS B
   WHERE B.col1 > A.col1) - 1 AS end_range
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.col1 = A.col1 + 1)
  AND col1 < (SELECT MAX(col1) FROM dbo.T1);
```

זוהי גישה אחת לפתרון הבעיה. גישה אחרת, אותה אני מוצא פשוטה ואינטואיטיבית בהרבה, היא הגישה הבאה:

- לכל ערך קיים, צרף את הערך הבא, ליצירת זוגות נוכחי, הבא.
- שמור רק זוגות בהן הבא פחות נוכחי גדול מאחד.
- עם הזוגות הנותרים, הוסף אחד לנוכחי והחסר אחד מהבא.

גישה זו נשענת על העובדה שערכים סמוכים עם מרחק גדול מאחד מייצגים את הגבולות של פער. זיהוי פער בהתבסס על זיהוי הערך הקיים הבא היא עוד שיטה שימושית.

לתרגום הצעדים לעיל ל– T-SQL, השאילתה הבאה פשוט מחזירה את הערך הבא לכל ערך נוכחי, ומייצרת את הפלט המוצג בטבלה 42–4:

```
SELECT col1 AS cur,
  (SELECT MIN(col1) FROM dbo.T1 AS B
   WHERE B.col1 > A.col1) AS nxt
FROM dbo.T1 AS A;
```

טבלה 42–4: זוגות ערכי נוכחי, הבא

cur	*nxt*
1	2
2	3
3	100
100	101
101	103
103	104
104	105
105	106
106	NULL

לבסוף, צור טבלה נגזרת מהשאילתה של השלב הקודם, ושמור רק זוגות בהם nxt – cur גדול מאחד. יש להוסיף אחד ל–cur להשגת התחלת הפער בפועל, ולהחסיר אחד מ–nxt להשגת סיום הפער בפועל:

```
SELECT cur + 1 AS start_range, nxt - 1 AS end_range
FROM (SELECT col1 AS cur,
         (SELECT MIN(col1) FROM dbo.T1 AS B
         WHERE B.col1 > A.col1) AS nxt
      FROM dbo.T1 AS A) AS D
WHERE nxt - cur > 1;
```

שים לב שפתרון זה נפטר מהנקודה לפני אינסוף ללא טיפול מיוחד, מכיוון שהערך nxt עבורה היה NULL.

השוואת הביצועים של שני הפתרונות מראה שהם דומים. למרות זאת, אין ספק שהפתרון השני פשוט ואינטואיטיבי יותר, וזהו יתרון גדול מבחינת קריאות ותחזוקה.

דרך אגב, אם עליך להחזיר את רשימת הערכים החסרים האינדיבידואליים בניגוד לטווחים חסרים, בעזרת שימוש בטבלה Nums המשימה פשוטה מאוד:

```
SELECT n FROM dbo.Nums
WHERE n BETWEEN (SELECT MIN(col1) FROM dbo.T1)
            AND (SELECT MAX(col1) FROM dbo.T1)
 AND NOT EXISTS (SELECT * FROM dbo.T1 WHERE col1 = n);
```

טווחים קיימים (ידועים גם כאיים)

החזרת טווחים של ערכים קיימים או קריסה של טווחים עם ערכים רציפים מערבת מושג שעוד לא נדון – גורם הקבצה. למעשה עליך לקבץ נתונים לפי גורם שאינו קיים בנתונים כמאפיין בסיס. במקרה שלנו, עליך לחשב ערך x כלשהו לכל האיברים של קבוצת הערכים הרציפים הראשונה $\{1, 2, 3\}$, ערך y כלשהו לשנייה $\{100, 101\}$, ערך z כלשהו לשלישית $\{104, 105, 106\}$ וכך הלאה. ברגע שבידך גורם הקבצה זה, באפשרותך לקבץ את הנתונים לפי גורם זה ולהחזיר את ערכי המינימום והמקסימום של col1 לכל קבוצה.

גישה אחת לחישוב גורם הקבצה זה מביאה אותי לשיטה נוספת: חישוב ערך min או max של קבוצת ערכים רציפים. קח את קבוצה $\{1, 2, 3\}$ כדוגמה. אם תצליח לחשב לכל אחד מהאיברים את הערך max בקבוצה (3), תוכל להשתמש בו כגורם ההקבצה שלך.

הלוגיקה מאחורי השיטה לחישוב המקסימום בתוך קבוצה של ערכים רציפים היא: החזר את הערך המינימלי הגדול מ או שווה לנוכחי, שאחריו יש פער. להלן התרגום ל–T-SQL, המפיק את הפלט המוצג בטבלה 4–43:

```
SELECT col1,
  (SELECT MIN(col1) FROM dbo.T1 AS B
   WHERE B.col1 >= A.col1
     AND NOT EXISTS
       (SELECT * FROM dbo.T1 AS C
        WHERE B.col1 = C.col1 - 1)) AS grp
FROM dbo.T1 AS A;
```

טבלה 4–43: גורם הקבצה

col1	*grp*
1	3
2	3
3	3
100	101
101	101
103	106
104	106
105	106
106	106

ההמשך ממש פשוט: צור טבלה נגזרת מתוך השאילתה של השלב הקודם, קבץ את הנתונים לפי גורם ההקבצה, והחזר את ערכי המינימום והמקסימום לכל קבוצה:

```
SELECT MIN(col1) AS start_range, MAX(col1) AS end_range
FROM (SELECT col1,
        (SELECT MIN(col1) FROM dbo.T1 AS B
         WHERE B.col1 >= A.col1
           AND NOT EXISTS
             (SELECT * FROM dbo.T1 AS C
              WHERE B.col1 = C.col1 - 1)) AS grp
      FROM dbo.T1 AS A) AS D
GROUP BY grp;
```

השאילתה פותרת את הבעיה, אך איני בטוח שהייתי מתאר אותה כפתרון פשוט ואינטואיטיבי מאוד בעל ביצועים משביעי רצון.

בחיפוש אחר דרך פשוטה ומהירה יותר לחישוב גורם ההקבצה, בחן את הפלט המוצג בטבלה 4–44 של השאילתה הבאה שמייצר פשוט מספרי שורה בהתבסס על מיון col1.

```
SELECT col1, ROW_NUMBER() OVER(ORDER BY col1) AS rn
FROM dbo.T1;
```

טבלה 4–44: מספרי שורה בהתבסס על מיון col1

col1	*rn*
1	1
2	2
3	3
100	4
101	5
103	6
104	7
105	8
106	9

אם אתה עובד עם SQL Server 2000, סביר שתרצה להשתמש בשיטה מבוססת–IDENTITY שתיארתי קודם לחישוב מספרי שורה.

ראה אם ביכולתך לזהות את הקשר בין הצורה בה ערכי col1 גדלים והצורה בה מספרי שורה גדלים.

ובכן, ההפרש נשאר קבוע בתוך אותה קבוצה של ערכים רציפים, מכיוון שאין בה פערים. ברגע שאתה מגיע לקבוצה חדשה, הפרש בין col1 ומספר השורה גדל.

ליתר המחשה של הרעיון, חשב את ההפרש בין col1 ומספר השורה ובחן את התוצאה המוצגת בטבלה 4–45. שיטת מפתח זו מראה דרך אחת לחישוב גורם ההקבצה עם ROW_NUMBER.

```
SELECT col1, col1 - ROW_NUMBER() OVER(ORDER BY col1) AS diff
FROM dbo.T1;
```

טבלה 4–45: הפרש בין col1 ומספר שורה

col1	*diff*
1	0
2	0
3	0
100	96
101	96
103	97
104	97
105	97
106	97

כעת זה ברור כשמש. יצרת גורם הקבצה במאמץ אפסי. פשוט ומהיר!

כעת פשוט החלף בפתרון הקודם את השאילתה של הטבלה הנגזרת עם הקוד הזה להשגת התוצאה המבוקשת:

```
SELECT MIN(col1) AS start_range, MAX(col1) AS end_range
FROM (SELECT col1, col1 - ROW_NUMBER() OVER(ORDER BY col1) AS grp
      FROM dbo.T1) AS D
GROUP BY grp;
```

סיכום

פרק זה כיסה נושאים רבים, כולם קשורים לתת–שאילתות. דנתי בתת–שאילתות סקלאריות ושל רשימה, תת–שאילתות עצמאיות ותלויות, ביטויי טבלה וחישובי דירוג. חשוב לזכור את שיטות היסוד שהצגתי כאן ושאציג לאורך כל הספר, כגון יצירת כפילויות בעזרת שימוש בטבלת עזר של מספרים, הצגת שובר–שוויון, מציאת נקודות לפני פער, החזרת הערך הבא או הקודם, חישוב גורם הקבצה וכד'. כל אלו בונים את אוצר המילים שלך ב– T-SQL ומשפרים את הכישורים שלך. ככל שתתקדם בגישה זו, תראה שזיהוי אלמנטים יסודיים בבעיה הופך קל יותר ויותר. לאחר שכבר פתרת ושייפת שיטות מפתח בנפרד בצורה ממוקדת, תשתמש בהן בצורה טבעית לפתרון בעיות.

5

Joins ופעולות סטים

פרק זה עוסק ב–joins ופעולות סטים – ההיבטים הלוגיים שלהם כמו גם ההיבטים הפיסיים והיבטי הביצועים. לכל סוג join ופעולת סטים אתן דוגמה ליישומים מעשיים. השתמשתי במינוח של ANSI SQL לצורך שיוך האלמנטים של השפה אותם אחלק כאן לקטגוריות. **Joins** (CROSS, INNER, OUTER) מתייחסים (אם נאמר בצורה כללית מעט) לפעולות **אופקיות** בין טבלאות, בעוד **פעולות סטים** (UNION, EXCEPT, INTERSECT) מתייחסות לפעולות **אנכיות** בין טבלאות.

Joins

Joins הן פעולות המאפשרות לך להתאים שורות בין טבלאות. אני מתייחס לפעולות אלו בצורה לא רשמית כפעולות אופקיות מכיוון שהטבלה הווירטואלית הנוצרת מפעולת join בין שתי טבלאות מכילה את כל הטורים משתי הטבלאות.

ראשית אתאר את צורות התחביר השונות של joins הנתמכות על ידי התקן, ואזכיר גם את האלמנטים הייחודיים ל–T-SQL. לאחר מכן אתאר את סוגי ה–joins הבסיסיים ואת היישומים שלהם ואמשיך בקטגוריות נוספות של joins. אקיים גם דיון ממוקד על העיבוד הפנימי של joins – בפרט, אלגוריתמים של joins.

יהיו לך כמה הזדמנויות לתרגל את מה שלמדת על ידי כך שתנסה לפתור בעיה המקיפה היבטים של joins שנדונו.

סגנון ישן לעומת סגנון חדש

T-SQL תומך בשתי צורות תחביר של joins. יש הרבה בלבול מסביב לשתיים. מתי להשתמש בכל אחת? לאיזו ביצועים טובים יותר? איזו שייכת לתקן ואיזו ייחודית ל–T-SQL? האם צורת התחביר הישנה תוסר מהמוצר בקרוב? וכו'. אני מקווה שפרק זה יסייע לפזר את הערפל.

אתחיל בכך שקיימות שתי צורות תחביר של joins הנתמכות על ידי תקן ANSI, ואף אחת מהן עדיין לא מתוכננת להיות מוסרת מהמוצר. האלמנטים של joins של התקן הישן יותר הם חלק בלתי נפרד מהתקן החדש. המשמעות היא שבאפשרותך להשתמש בכל אחת מהן מבלי לדאוג שהן לא תיתמכנה על ידי Microsoft SQL Server בקרוב. SQL Server לא יסיר תמיכה מאלמנטים שעדיין נתמכים על ידי התקן.

הצורה הישנה יותר מבין שתי צורות התחביר הוצגה בתקן ANSI SQL:1989. מה שמבדיל אותה מהצורה החדשה, הוא השימוש בפסיקים להפרדה בין שמות טבלאות המופיעים בפסוקית FROM, והיעדר מילת המפתח JOIN והפסוקית ON:

```
FROM T1, T2
WHERE where_filter
```

התקן ANSI SQL:1989 תמך רק בסוגים cross join ו– inner join. הוא לא תמך ב– outer joins.

צורת התחביר החדשה הוצגה בתקן ANSI SQL:1992 ומה שמבדיל אותה מצורת התחביר הישנה, היא ההסרה של הפסיקים וההוספה של מילת המפתח JOIN ושל הפסוקית ON:

```
FROM T1 <join_type> JOIN T2 ON <on_filter>
WHERE where_filter
```

התקן ANSI SQL:1992 מוסיף תמיכה ב– outer joins, דבר אשר גורר את הצורך להפרדת מסננים – המסנן ON והמסנן WHERE. אסביר זאת בפירוט בסעיף outer joins.

חלק מהבלבול סביב שתי צורות התחביר קשור לעובדה ש–T-SQL תמך בתחביר ייחודי עבור outer joins בטרם SQL Server הוסיף תמיכה בתחביר של ANSI SQL:1992. היה צורך מעשי ב– outer joins, ו– SQL Server סיפק תשובה לצורך זה. במיוחד אני מתכוון לתחביר הייחודי בסגנון–הישן של outer join המשתמש ב– =* וב– *= עבור left outer join ועבור right outer join, בהתאמה.

מסיבות של תאימות אחורה, SQL Server לא הסיר עד כה תמיכה לתחביר הייחודי של outer join. עם זאת, אלמנטים תחביריים אלו הוסרו מהמוצר ב– SQL Server 2005 ויעבדו רק תחת דגלון של תאימות–אחורה. כל האלמנטים התחביריים האחרים של joins הם בתקן ואינם מועמדים להסרה – לא על ידי התקן ולא על ידי SQL Server.

במהלך ההסבר על הסוגים הבסיסיים השונים של joins, אדון בשתי צורות התחביר ואציג את דעתי בסוגיה איזו מהן אני מוצא נוחה יותר לשימוש ומדוע.

סוגי Join בסיסיים

במהלך ההסבר על הסוגים הבסיסיים השונים של joins – cross, inner ו–outer – זכור את השלבים של עיבוד לוגי של שאילתות אותם הצגתי בפירוט בפרק 1. במיוחד, זכור את השלבים הלוגיים המעורבים בעיבוד של join.

כל סוג בסיסי של join מתרחש רק בין שתי טבלאות. אפילו אם יש לך יותר משתי טבלאות בפסוקית FROM, שלושת הצעדים הראשונים של עיבוד לוגי של שאילתה יתרחשו בין שתי טבלאות בכל פעם. כל join יפיק טבלה וירטואלית, שבתורה תבצע join עם הטבלה הבאה בפסוקית FROM. התהליך ימשיך עד שיעובדו כל הטבלאות בפסוקית FROM.

סוגי ה–join הבסיסיים נבדלים בשלבים הלוגיים שהם מפעילים. cross join מפעיל רק את הראשון (מכפלה קרטזית), inner join מפעיל את הראשון ואת השני (מכפלה קרטזית, מסנן ON), ו– outer join מפעיל את כל השלושה (מכפלה קרטזית, מסנן ON, הוספת שורות חיצוניות).

Cross Join

cross join מבצע מכפלה קרטזית בין שתי טבלאות. במילים אחרות, הוא מחזיר שורה לכל שילוב אפשרי של שורה מהטבלה השמאלית ושורה מהטבלה הימנית. אם לטבלה השמאלית קיימות n שורות, ולטבלה הימנית קיימות m שורות, cross join יחזיר טבלה בת n x m שורות.

קיימים הרבה יישומים מעשיים ל– cross joins, אך אתחיל בדוגמה מאוד פשוטה.

השאילתה הבאה מפיקה את כל הזוגות האפשריים של עובדים מטבלת Employees במסד הנתונים Northwind:

```
USE Northwind;

SELECT E1.FirstName, E1.LastName AS emp1,
  E2.FirstName, E2.LastName AS emp2
FROM dbo.Employees AS E1
  CROSS JOIN dbo.Employees AS E2;
```

מכיוון שטבלת Employees מכילה תשע שורות, סט התוצאה יכיל 81 שורות.

ולהלן התחביר של ANSI SQL:1989 בו תשתמש לאותה מטרה:

```
SELECT E1.FirstName, E1.LastName AS emp1,
  E2.FirstName, E2.LastName AS emp2
FROM dbo.Employees AS E1, dbo.Employees AS E2;
```

עבור cross joins בלבד, אני מעדיף להשתמש בפסיק (בניגוד לשימוש במילות המפתח CROSS JOIN) מכיוון שהדבר מאפשר קוד קצר יותר. אני מוצא גם שהתחביר הישן טבעי וקריא יותר. ה–optimizer יפיק את אותה תוכנית עבור שתיהן, כך שאינך צריך לדאוג בנוגע לביצועים.

כפי שתראה בהמשך, אתן המלצה שונה בנוגע ל– inner joins. כעת הבה נתבונן בשימושים מתוחכמים יותר של cross joins.

בפרק 4 הצגתי שיטת מפתח חזקה מאוד לייצור עותקים של שורות. זכור שהשתמשתי בטבלת עזר של מספרים (Nums) כלהלן כדי לייצר את המספר המבוקש של עותקים לכל שורה:

```
SELECT ...
FROM T1, Nums
WHERE n <= <num_of_dups>
```

השיטה לעיל תייצר בסט התוצאה num_of_dups עותקים של כל שורה ב–T1. כדוגמה מעשית, נניח שעליך למלא טבלת Orders עם נתוני דוגמה לצורך בדיקות. יש לך טבלת Customers עם מידע לקוחות לדוגמה וטבלת Employees עם מידע עובדים לדוגמה. ברצונך לייצר, לכל שילוב של לקוח ועובד, הזמנה לכל יום בינואר 2006.

אדגים שיטה זו – ייצור נתונים לדוגמה בהתבסס על שכפול שורות, במסד הנתונים Northwind. טבלת Customers מכילה 91 שורות, טבלת Employees מכילה 9 שורות, ולכל שילוב לקוח–עובד, אתה צריך הזמנה לכל יום בינואר 2006 – כלומר, עבור 31 ימים. סט התוצאה אמור להכיל 25,389 שורות (91 x 9 x 31 = 25,389). באופן טבעי, תרצה לאחסן את סט התוצאה בטבלת יעד ולייצר קוד הזמנה לכל הזמנה.

קיימות לך כבר טבלאות לקוחות ועובדים, אך חסרה טבלה – אתה צריך טבלה המייצגת את הימים. בטח ניחשת כבר שטבלת Nums תקבל את תפקיד הטבלה החסרה:

```
SELECT CustomerID, EmployeeID,
  DATEADD(day, n-1, '20060101') AS OrderDate
FROM dbo.Customers, dbo.Employees, dbo.Nums
WHERE n <= 31;
```

אתה מצליב את הטבלאות Customers, Employees ו–Nums, תוך שאתה מסנן את 31 הערכים הראשונים של n מטבלת Nums עבור 31 הימים בחודש. ברשימת ה–SELECT, אתה מחשב את תאריכי היעד הספציפיים על ידי הוספת n-1 ימים ליום הראשון של החודש, 1 בינואר, 2006.

האלמנט האחרון שחסר הוא קוד ההזמנה. אך תוכל לייצר אותו בקלות על ידי שימוש בפונקציה ROW_NUMBER ב– SQL Server 2005 או בפונקציה או התכונה IDENTITY ב– SQL Server 2000.

בפועל סביר להניח שתרצה לעטוף את הלוגיקה הזו בפרוצדורה מאוחסנת המקבלת את תחום התאריכים כקלט. במקום להשתמש בקבוע עבור מספר הימים במסנן, תשתמש בביטוי הבא:

```
DATEDIFF(day, @fromdate, @todate) + 1
```

בדומה, הפונקציה DATEADD ברשימת ה–SELECT תתייחס ל–@fromdate במקום התאריך הקבוע:

```
DATEADD(day, n-1, @fromdate) AS OrderDate
```

להלן הקוד לו תידרש ב– SQL Server 2000 לייצור נתוני הדוגמה למילוי טבלת יעד. אני משתמש בפונקציה IDENTITY לייצור קודי ההזמנה ובגבולות תחום התאריכים כארגומנטים בקלט:

```
DECLARE @fromdate AS DATETIME, @todate AS DATETIME;
SET @fromdate = '20060101';
SET @todate = '20060131';

SELECT IDENTITY(int, 1, 1) AS OrderID,
  CustomerID, EmployeeID,
  DATEADD(day, n-1, @fromdate) AS OrderDate
INTO dbo.MyOrders
FROM dbo.Customers, dbo.Employees, dbo.Nums
WHERE n <= DATEDIFF(day, @fromdate, @todate) + 1;
```

ב– SQL Server 2005, תוכל להשתמש בפונקציה ROW_NUMBER במקום בפונקציה IDENTITY, ולשייך את מספרי השורה בהתבסס על טור מיון רצוי (למשל, OrderDate):

```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO

DECLARE @fromdate AS DATETIME, @todate AS DATETIME;
SET @fromdate = '20060101';
SET @todate = '20060131';

WITH Orders
AS
(
  SELECT CustomerID, EmployeeID,
    DATEADD(day, n-1, @fromdate) AS OrderDate
  FROM dbo.Customers, dbo.Employees, dbo.Nums
  WHERE n <= DATEDIFF(day, @fromdate, @todate) + 1
)
SELECT ROW_NUMBER() OVER(ORDER BY OrderDate) AS OrderID,
  CustomerID, EmployeeID, OrderDate
INTO dbo.MyOrders
FROM Orders;
```

כאשר תסיים להתנסות בקוד זה, אל תשכח להסיר את טבלת MyOrders:

```
DROP TABLE dbo.MyOrders;
```

יישום נוסף של cross join מאפשר לך לשפר ביצועים של שאילתות המפעילות חישובים בין מאפייני שורה לבין צבירות על פני שורות. כדי להדגים שיטת מפתח זו, אשתמש בטבלת sales במסד הנתונים pubs. ראשית, צור אינדקס על הטור qty, החשוב למשימה שלפנינו:

```
USE pubs;
CREATE INDEX idx_qty ON dbo.sales(qty);
```

המשימה שלפנינו היא לחשב עבור כל מכירה את אחוז המכירה מסך הכמות שנמכרה, ואת ההפרש בין כמות המכירה לכמות הממוצעת של כל המכירות. הדרך האינטואיטיבית לתוכניתנים לכתוב חישובים בין מאפייני שורה לבין צבירות על פני שורות היא להשתמש בתת–שאילתות. הקוד הבא (המפיק את הפלט המוצג בטבלה 1–5) מדגים את שיטת תת–השאילתות:

```
SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / (SELECT SUM(qty) FROM dbo.sales) * 100
       AS DECIMAL(5, 2)) AS per,
  qty - (SELECT AVG(qty) FROM dbo.sales) as diff
FROM dbo.sales;
```

טבלה 1–5: נתוני מכירות, כולל אחוז מסך הכל והפרש מממוצע

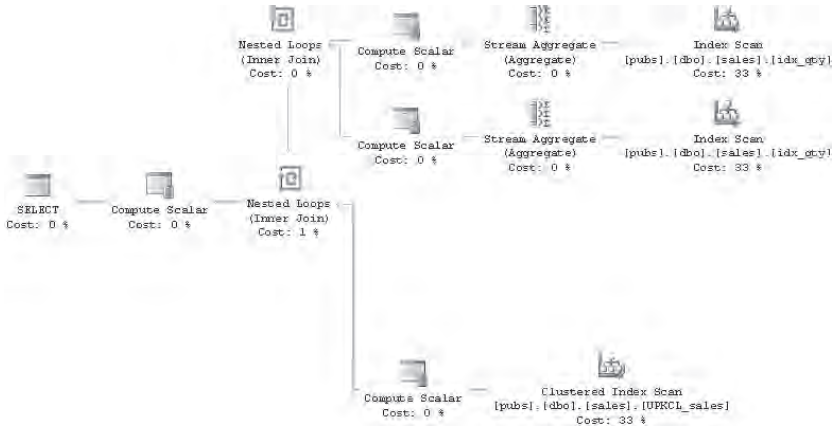
stor_id	*ord_num*	*title_id*	*ord_date*	*qty*	*per*	*diff*
6380	6871	BU1032	1994-09-14	5	1.01	-18
6380	722a	PS2091	1994-09-13	3	0.61	-20
7066	A2976	PC8888	1993-05-24	50	10.14	27
7066	QA7442.3	PS2091	1994-09-13	75	15.21	52
7067	D4482	PS2091	1994-09-14	10	2.03	-13
7067	P2121	TC3218	1992-06-15	40	8.11	17
7067	P2121	TC4203	1992-06-15	20	4.06	-3
7067	P2121	TC7777	1992-06-15	20	4.06	-3
7131	N914008	PS2091	1994-09-14	20	4.06	-3
7131	N914014	MC3021	1994-09-14	25	5.07	2
7131	P3087a	PS1372	1993-05-29	20	4.06	-3
7131	P3087a	PS2106	1993-05-29	25	5.07	2

stor_id	*ord_num*	*title_id*	*ord_date*	*qty*	*per*	*diff*
7131	P3087a	PS3333	1993-05-29	15	3.04	-8
7131	P3087a	PS7777	1993-05-29	25	5.07	2
7896	QQ2299	BU7832	1993-10-28	15	3.04	-8
7896	TQ456	MC2222	1993-12-12	10	2.03	-13
7896	X999	BU2075	1993-02-21	35	7.10	12
8042	423LL922	MC3021	1994-09-14	15	3.04	-8
8042	423LL930	BU1032	1994-09-14	10	2.03	-13
8042	P723	BU1111	1993-03-11	25	5.07	2
8042	QA879.1	PC1035	1993-05-22	30	6.09	7

בטרם אכנס להיבטי הביצועים של השאילתה, ארצה ראשית לדון בכמה מההיבטים הלוגיים שלה. כפי שתוכל לראות, הן הכמות הכוללת והן הכמות הממוצעת מושגות על ידי שימוש בתת–שאילתות עצמאיות. הביטוי המחשב את האחוז, לוקח בחשבון את הדרך בה ביטויים מעובדים ב–T-SQL. טיפוס נתונים של ביטוי נקבע על ידי טיפוס הנתונים עם הקדימות הגבוהה יותר בין האופרנדים. המשמעות היא ש– qty / totalqty (כאשר totalqty מייצגת את תת–השאילתה המחזירה את הכמות הכוללת) תניב כתוצאה מספר שלם – מכיוון ששני האופרנדים הם מספרים שלמים, והאופרטור / יהיה חלוקת שלמים. מכיוון ש–qty היא חלק של totalqty, תמיד תקבל אפס בתוצאה, מכיוון שהשארית נקטעת. כדי לקבל חישוב עשרוני מדויק, תצטרך להמיר את האופרנדים למספרים עשרוניים. זאת ניתן להשיג על ידי שימוש בהמרה ישירה או בהמרה עקיפה, כפי שאני עשיתי. 1.*qty ימיר בעקיפין את הערך qty למספר עשרוני מכיוון שמספר עשרוני גבוה בקדימות מאשר מספר שלם. כתוצאה מכך, totalqty גם היא תומר למספר עשרוני, וכך גם המספר השלם 100. לבסוף, השתמשתי בהמרה ישירה ל– DECIMAL(5, 2) עבור טור התוצאה per כדי לשמור על דיוק של שני מקומות עשרוניים בלבד. שלוש ספרות לשמאל הנקודה העשרונית מספיקות כאן עבור האחוזים מכיוון שהערך המרבי האפשרי הוא 100.

באשר לביצועים, בחן את תוכנית העבודה של השאילתה הזו, המוצגת בתרשים 1–5.

תרשים 1–5: תוכנית עבודה להשגת צבירות עם תת–שאילתות

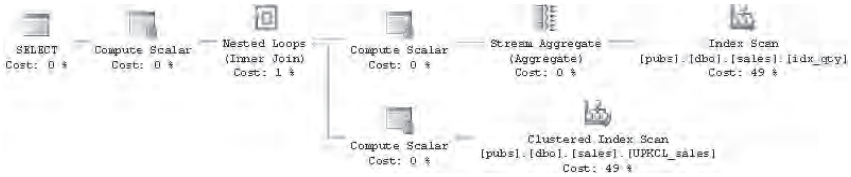


שים לב שהאינדקס שייצרתי על הטור qty נסרק פעמיים – פעם אחת לחישוב הסכום, ופעם אחת לחישוב הממוצע. במילים אחרות, בהנחה שיש לך אינדקס על טור הצבירה, האינדקס ייסרק פעם אחת לכל תת–שאילתה המחזירה צבירה. אם אין לך אינדקס המכיל את טור הצבירה, העניינים אף גרועים יותר, שכן תקבל סריקת טבלה לכל תת–שאילתה.

ניתן לבצע אופטימיזציה על שאילתה זו על ידי שימוש בשיטת מפתח המשתמשת ב– cross join. תוכל לחשב את כל הצבירות הנדרשות בשאילתה אחת, שתדרוש סריקת אינדקס יחידה בלבד או סריקת טבלה יחידה. שאילתה כזו תפיק שורת תוצאה אחת לכל הצבירות. אתה יוצר טבלה נגזרת המוגדרת על ידי שאילתה זו ומצליב אותה עם טבלת הבסיס. כעת יש לך גישה הן למאפייני הבסיס והן לצבירות. להלן שאילתת הפתרון, המייצרת את התוכנית האופטימלית יותר המוצגת בתרשים 2–5.

```
SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / sumqty * 100 AS DECIMAL(5, 2)) AS per,
  qty - avgqty as diff
FROM dbo.sales,
  (SELECT SUM(qty) AS sumqty, AVG(qty) AS avgqty
   FROM dbo.sales) AS AGG;
```

תרשים 2–5: תוכנית עבודה להשגת צבירות בעזרת cross join



כפי שתוכל לראות בתוכנית, האינדקס על הטור qty נסרק רק פעם אחת, ושתי הצבירות מחושבות באותה סריקה. כמובן, ב– SQL Server 2005 תוכל להשתמש בביטוי טבלה שגור (CTE), שייתכן שתמצא אותו קל יותר לקריאה:

```
WITH Agg AS
(
  SELECT SUM(qty) AS sumqty, AVG(qty) AS avgqty
  FROM dbo.sales
)
SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / sumqty * 100 AS DECIMAL(5, 2)) AS per,
  qty - avgqty as diff
FROM dbo.sales, Agg;
```

תמצא ששתי השאילתות מייצרות את אותה תוכנית. בפרק 6 אדגים כיצד להשתמש בפסוקית OVER החדשה ב– SQL Server 2005 כדי להתמודד עם בעיות דומות.

כשתסיים להתנסות בשיטה זו, הסר את האינדקס על הטור qty:

```
DROP INDEX dbo.sales.idx_qty;
```

Inner Join

inner joins משמשים להתאמת שורות בין שתי טבלאות בהתבסס על קריטריון כלשהו. מתוך שלושת השלבים הראשונים של עיבוד לוגי של שאילתה, inner joins מפעילים את השניים הראשונים – מכפלה קרטזית והמסנן ON. אין שלב המוסיף שורות חיצוניות. כתוצאה מכך, בשאילתת INNER JOIN המכילה הן פסוקית ON והן פסוקית WHERE, לוגית הפסוקיות מופעלות זו אחר זו. למעט יוצא מהכלל אחד – אין הבדל בין הגדרת ביטוי לוגי בפסוקית ON או בפסוקית WHERE של INNER JOIN – מכיוון שאין בין השתיים שלב ביניים המוסיף שורות חיצוניות.

היוצא מהכלל היחיד הוא כאשר אתה מגדיר GROUP BY ALL. זכור ש– GROUP BY ALL מחזיר קבוצות שסוננו החוצה על ידי הפסוקית WHERE, אך הוא אינו מחזיר קבוצות שסוננו החוצה על ידי הפסוקית ON. זכור גם שזהו ביטוי לא סטנדרטי שנשאר במוצר מסיבות היסטוריות ושעליך להימנע משימוש בו.

באשר לביצועים, כאשר אינך משתמש באפשרות GROUP BY ALL, לרוב תקבל את אותה תוכנית ללא קשר למיקום ביטוי המסנן. זאת מכיוון שה–optimizer מודע לכך שאין הבדל. עלי להיזהר תמיד כשאני אומר דברים כאלו בהקשר לבחירות אופטימיזציה, מכיוון שהתהליך הוא כה דינמי.

באשר לשתי צורות התחביר הנתמכות, על ידי שימוש בתחביר של ANSI SQL:1992, יש לך גמישות רבה יותר בבחירת הפסוקית שבה תשתמש להגדרת ביטוי המסנן שלך. מכיוון שלוגית אין הבדל היכן אתה ממקם את המסנן שלך, ולרוב אין גם הבדל ביצועים. הקווים המנחים שלך צריכים להיות כתיבה טבעית ואינטואיטיבית. כתוב בצורה המרגישה טבעית יותר עבורך ועבור התוכניתנים הנדרשים לתחזק את הקוד שלך. למשל, עבורי מסנן המתאים מאפיינים בין הטבלאות צריך להופיע בפסוקית ON, בעוד מסנן על מאפיינים מטבלה אחת בלבד צריך להופיע בפסוקית WHERE. אשתמש בשאילתה הבאה כדי להחזיר הזמנות שבוצעו על ידי לקוחות מארה"ב:

```
USE Northwind;

SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID
WHERE Country = 'USA';
```

כאשר אתה משתמש בתחביר של ANSI SQL:1989, אין לך ברירה אלא להגדיר את כל ביטויי המסנן בפסוקית WHERE:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID = O.CustomerID
  AND Country = 'USA';
```

זכור שהדיון כאן הוא בנוגע ל– inner joins; בנוגע ל– outer joins, קיימים הבדלים לוגיים בין הגדרת ביטוי מסנן בפסוקית ON לבין הגדרתו בפסוקית WHERE.

הזכרתי קודם שאני אוהב להשתמש בתחביר של ANSI SQL:1989 עבור cross joins. עם זאת, לגבי inner joins, ההמלצה שלי שונה. הסיבה לכך היא שקיים סיכון בשימוש בתחביר של ANSI SQL:1989.

אם אתה שוכח להגדיר את תנאי ה–join, בלי כוונה אתה מקבל cross join, כפי שניתן לראות בקוד הבא:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O;
```

ב– SSMS (SQL Server Management Studio), תוכנית השאילתה עבור cross join תכלול אופרטור join המסומן בסימן אזהרה צהוב, וההודעה שתצוץ בחלק ה–Warnings תאמר "No Join Predicate". אזהרה זו עוצבה כדי להזכיר לך שייתכן ששכחת להגדיר ביטוי join.

עם זאת, אם אתה מגדיר במפורש INNER JOIN, כאשר אתה כותב שאילתת inner join, נדרשת פסוקית ON. אם שכחת להגדיר את כל תנאי join, ה–parser יתפוס את הטעות והשאילתה לא תרוץ:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C JOIN dbo.Orders AS O;

Msg 102, Level 15, State 1, Line 2
Incorrect syntax near ';'.
```

ה–parser מצא סימן נקודה–פסיק אחרי dbo.Orders AS O, למרות שהוא מצפה למשהו אחר (פסוקית ON או אפשרויות אחרות), לכן הוא מייצר הודעת שגיאה אשר מודיעה שישנו תחביר שגוי ליד ';'.

שים לב: אם יש לך composite join (join המבוסס על מספר מאפיינים), ואתה מגדיר לפחות ביטוי אחד אך שוכח את האחרים, אף אחת מצורות התחביר לא תתפוס את השגיאה. בדומה, שגיאות לוגיות אחרות לא ייתפסו – למשל, אם אתה מקליד בטעות ON C.OrderID = C.OrderID.

Outer Join

outer joins משמשים להחזרת שורות תואמות משתי הטבלאות בהתבסס על קריטריון כלשהו, וכן שורות מהטבלה ה"שמורה" (או טבלאות) שעבורן לא נמצאו התאמות.

אתה מסמן טבלאות שמורות על ידי מילות המפתח LEFT, RIGHT או FULL. LEFT מסמן את הטבלה השמאלית כשמורה, RIGHT מסמן את הימנית ו–FULL מסמן את שתיהן.

outer joins מפעילים את כל שלושת שלבי העיבוד הלוגי של שאילתה – מכפלה קרטזית, מסנן ON והוספת שורות חיצוניות. שורות חיצוניות המתווספות עבור שורות מהטבלה השמורה שלא מצאו התאמה, מקבלות NULLs עבור המאפיינים של הטבלה הלא–שמורה.

השאילתה הבאה מחזירה לקוחות עם קודי ההזמנה שלהם (ממש כפי שיעשה inner join עם אותה פסוקית ON), אך היא מחזירה גם שורה לכל לקוח שאין לו הזמנות מכיוון שמילת המפתח LEFT מגדירה את הטבלה Customers כשמורה:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID;
```

מילת המפתח OUTER אופציונלית מכיוון שהזכרת אחת ממילות המפתח LEFT, RIGHT או FULL מרמזת על outer join. עם זאת, שלא כמו inner joins, בהם מרבית התוכניתנים לא מציינים את מילת המפתח האופציונלית INNER, מרבית התוכניתנים (ואני ביניהם) לרוב כן מציינים את מילת המפתח OUTER. אני מניח שזה מרגיש יותר טבעי.

כפי שציינתי קודם, SQL Server 2005 יתמוך בתחביר הייחודי שאינו בתקן עבור outer joins רק תחת דגלון של תאימות–אחורה. כדי לאפשר את התחביר הישן, שנה את מצב התאימות של מסד הנתונים Northwind ל–80 (SQL Server 2000):

```
EXEC sp_dbcmptlevel Northwind, 80;
```

ה– outer join בתחביר הישן צוין בפסוקית WHERE, ולא בפסוקית FROM. במקום =, הוא השתמש ב– =* כדי לייצג left outer join וב– *= כדי לייצג right outer join. לא הייתה תמיכה ב– full outer join. למשל, השאילתה הבאה מחזירה לקוחות עם קודי ההזמנה שלהם, ולקוחות ללא הזמנות:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID *= O.CustomerID;
```

תחביר זה בעייתי מאוד מכיוון שלא קיימת הפרדה בין מסנן ON לבין המסנן WHERE. למשל, אם אתה מעוניין להחזיר רק לקוחות שאין להם הזמנות, בעזרת שימוש בתחביר ANSI הדבר פשוט מאוד:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID
WHERE O.CustomerID IS NULL;
```

אתה מקבל חזרה את הלקוחות FISSA ו–PARIS. השאילתה מפעילה תחילה את שלושת השלבים בעיבוד הלוגי של שאילתה, ומניבה טבלת ביניים וירטואלית המכילה לקוחות עם ההזמנות שלהם (שורות פנימיות) וגם לקוחות ללא הזמנות (שורות חיצוניות). עבור השורות החיצוניות, המאפיינים מטבלת Orders הם NULL. לאחר מכן מופעל המסנן

WHERE על תוצאת ביניים זו. רק השורות עם NULL בטור ה–join מהצד הלא–שמור, המייצגות לקוחות שאין להם הזמנות, ממלאות אחר התנאי בפסוקית WHERE.

אם תנסה לכתוב שאילתה זו על ידי שימוש בתחביר הישן, תקבל תוצאה מפתיעה:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID *= O.CustomerID
  AND O.CustomerID IS NULL;
```

השאילתה מחזירה את כל 91 הלקוחות. מכיוון שאין הבחנה בין פסוקית ON לפסוקית WHERE, הגדרתי את שני הביטויים בפסוקית WHERE כמופרדים על ידי האופרטור הלוגי AND. אינך יודע איזה חלק של המסנן יתרחש לפני הוספת השורות החיצוניות ואיזה חלק יתרחש לאחריה. זוהי בחירה בלעדית של SQL Server. אם תתבונן בתוצאה, תוכל לנחש מה SQL Server עשה. לוגית, הוא הפעיל את כל הביטוי לפני שהוסיף את השורות החיצוניות. כמובן, אין אף שורה במכפלה הקרטזית שעבורה **C.CustomerID = O.CustomerID** ו– **O.CustomerID IS NULL**, כך שהשלב השני בעיבוד הלוגי של השאילתה יניב סט ריק. השלב השלישי מוסיף שורות חיצוניות עבור שורות מהטבלה השמורה (Customers) שעבורן לא נמצאה התאמה. זאת הסיבה לכך שהשאילתה החזירה את כל 91 הלקוחות.

מתוך כך שאתה זוכר שקיבלת את התוצאה המפתיעה הזו, מכיוון ששני הביטויים הופעלו לפני הוספת השורות החיצוניות, ובהכירך את העיבוד הלוגי של שאילתה, קיימת דרך "לתקן" את הבעיה. זכור שקיים מסנן נוסף הזמין לך בשאילתה – המסנן HAVING. ראשית כתוב שאילתת join ללא המסנן המבודד שורות חיצוניות. למרות שהשורות בתוצאה של ה–join הן ייחודיות, קבץ אותן לפי כל הטורים כדי לאפשר לשאילתה לכלול פסוקית HAVING. אז הוסף את המסנן שמבודד את השורות החיצוניות בפסוקית HAVING:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID *= O.CustomerID
GROUP BY C.CustomerID, CompanyName, OrderID
HAVING OrderID IS NULL;
```

בצורה כזו אתה קובע איזה מסנן יופעל לפני הוספת שורות חיצוניות ואיזה אחרי.

שים לב: זכור שהדגמתי את התחביר הישן הייחודי ל–T–SQL רק כדי שתהיה מודע לסוגיות אלו. אני כמובן ממליץ בחום להימנע משימוש בו ולשנות את כל הקוד המשתמש בו כעת לתחביר ANSI. בקיצור, אל תנסה את זה בבית!

כאשר תסיים להתנסות בתחביר הישן, שנה את רמת התאימות של מסד הנתונים חזרה ל–90 (SQL Server 2005):

```
EXEC sp_dbcmptlevel Northwind, 90;
```

בפרק הקודם סיפקתי פתרון המשתמש בתת–שאילתות לבעיית הערך החסר הנמוך ביותר. כתזכורת, אתה עובד עם טבלה T1, אותה אתה יוצר וממלא על ידי הרצת הקוד בקטע–קוד 1–5.

קטע–קוד 1–5: יצירה ומילוי של טבלה T1

```
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  keycol  INT        NOT NULL PRIMARY KEY,
  datacol VARCHAR(10)NOT NULL
);
INSERT INTO dbo.T1(keycol, datacol) VALUES(1, 'e');
INSERT INTO dbo.T1(keycol, datacol) VALUES(2, 'f');
INSERT INTO dbo.T1(keycol, datacol) VALUES(3, 'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(4, 'b');
INSERT INTO dbo.T1(keycol, datacol) VALUES(6, 'c');
INSERT INTO dbo.T1(keycol, datacol) VALUES(7, 'd');
```

המשימה שלך היא למצוא את המפתח החסר המינימלי (במקרה זה, 5) בהנחה שהמפתח מתחיל ב–1. סיפקתי את הפתרון הבא המבוסס על תת–שאילתות:

```
SELECT MIN(A.keycol + 1)
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.keycol = A.keycol + 1);
```

זכור שסיפקתי ביטוי CASE המחזיר את הערך 1 אם הוא חסר; אחרת, הוא מחזיר את התוצאה של השאילתה הקודמת. תוכל לפתור את אותה בעיה – החזרת המפתח החסר המינימלי כאשר 1 קיים בטבלה – על ידי שימוש בשאילתת outer join בין שני מופעים של T1 כלהלן:

```
SELECT MIN(A.keycol + 1)
FROM dbo.T1 AS A
  LEFT OUTER JOIN dbo.T1 AS B
    ON B.keycol = A.keycol + 1
WHERE B.keycol IS NULL;
```

הצעד הראשון בפתרון הוא הפעלת ה– left outer join בין שני מופעים של T1, הנקראים A ו–B, בהתבסס על תנאי ה– join: B.keycol = A.keycol + 1. צעד זה מערב את שלושת השלבים הראשונים של עיבוד לוגי של שאילתה שתיארתי בפרק 1 (מכפלה קרטזית, מסנן ON והוספת שורות חיצוניות). כרגע, התעלם מהמסנן WHERE ומהפסוקית SELECT. תנאי ה–join מתאים לכל שורה מ–A, שורה מ–B שערך המפתח שלה גדול ב–1 מערך המפתח של A. מכיוון שזהו outer join, שורות מ–A שאין להן התאמה ב–B נוספות כשורות חיצוניות, ונוצרת הטבלה הווירטואלית המוצגת בטבלה 2–5.

טבלה 2–5: פלט של שלב 1 בפתרון ערך חסר מינימלי

A.keycol	*A.datacol*	*B.keycol*	*B.datacol*
1	e	2	f
2	f	3	a
3	a	4	b
4	b	NULL	NULL
6	c	7	d
7	d	NULL	NULL

שים לב שהשורות החיצוניות מייצגות את הנקודות לפני הפערים, מכיוון שחסר ערך המפתח הבא. השלב השני בפתרון הוא לבודד רק את הנקודות לפני הפערים, הפסוקית WHERE מסננת רק שורות עבורן B.keycol הוא NULL, ומייצרת את הטבלה הווירטואלית המוצגת בטבלה 3–5.

טבלה 3–5: פלט של שלב 2 בפתרון ערך חסר מינימלי

A.keycol	*A.datacol*	*B.keycol*	*B.datacol*
4	b	NULL	NULL
7	d	NULL	NULL

לבסוף, הצעד האחרון בפתרון הוא לבודד את ערך ה–A.keycol המינימלי, שהוא ערך המפתח המינימלי לפני פער, ולהוסיף 1. התוצאה היא הערך החסר המינימלי המבוקש.

ה–optimizer מייצר לשתי השאילתות תוכניות מאוד דומות, עם עלויות זהות. כך שבאפשרותך להשתמש בפתרון שאיתו אתה מרגיש יותר נוח. עבורי, הפתרון המבוסס על תת–שאילתות נראה יותר אינטואיטיבי.

סוגי Joins שאינם נתמכים

ANSI SQL תומך בשני סוגי join שאינם נתמכים על ידי T-SQL – natural join ו– union join. לא מצאתי יישומים מעשיים עבור union join, כך שלא אטרח לתאר או להדגים אותו בספר זה.

natural join הוא inner join שבו תנאי ה–join מבוסס בעקיפין על טורים תואמים בעלי אותו שם בשתי הטבלאות. התחביר עבור natural join, שלא במפתיע, הוא NATURAL JOIN. למשל, שתי השאילתות הבאות זהות לוגית, אך רק השנייה מזוהה על ידי SQL Server:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C NATURAL JOIN dbo.Orders AS O;
```

ו–

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON O.CustomerID = O.CustomerID;
```

דוגמאות נוספות של Joins

עד כה הצגתי סוגי join בסיסיים. קיימות דרכים אחרות לשייך joins לקטגוריות מלבד הסוג הבסיסי שלהם. בסעיף זה אתאר self joins, nonequijoins, שאילתות עם מספר joins ו– semi joins.

Self Join

self join הוא פשוט join בין שני מופעים של אותה טבלה. הראיתי כבר דוגמאות של self joins מבלי שסיווגתי אותם מפורשות ככאלה.

להלן דוגמה פשוטה של self join בין שני מופעים של טבלת Employees; אחד מייצג עובדים (E), והשני מייצג מנהלים (M):

```
USE Northwind;

SELECT E.FirstName, E.LastName AS emp,
  M.FirstName, M.LastName AS mgr
FROM dbo.Employees AS E
```

```
LEFT OUTER JOIN dbo.Employees AS M
    ON E.ReportsTo = M.EmployeeID;
```

השאילתה מייצרת את הפלט המוצג בטבלה 4–5, בה שמות העובדים מוחזרים עם שמות המנהלים שלהם.

טבלה 4–5: עובדים והמנהלים שלהם

emp	*mgr*
Nancy Davolio	Andrew Fuller
Andrew Fuller	NULL
Janet Leverling	Andrew Fuller
Margaret Peacock	Andrew Fuller
Steven Buchanan	Andrew Fuller
Michael Suyama	Steven Buchanan
Robert King	Steven Buchanan
Laura Callahan	Andrew Fuller
Anne Dodsworth	Steven Buchanan

השתמשתי ב– left outer join כדי לכלול בתוצאה את Andrew – סגן הנשיא של מכירות. יש לו NULL בטור ReportsTo מכיוון שאין לו מנהל.

שים לב: כאשר מחברים שני מופעים של אותה טבלה, חייבים לתת כינוי לפחות לאחת הטבלאות. דבר זה מספק שם או כינוי ייחודי לכל מופע, כך שאין רב–משמעיות בשמות טורי התוצאה ובשמות הטורים בטבלת הביניים הווירטואלית.

Nonequijoin

equijoins הם joins עם תנאי join המתבסס על אופרטור שוויון. ל–Nonequijoins יש אופרטור שונה משוויון בתנאי ה–join שלהם.

כדוגמה, נניח שעליך לייצר את כל הזוגות של שני עובדים שונים מטבלת Employees. הנח שכעת הטבלה מכילה קודי עובד A, B ו– C. cross join ייצר את תשעת הזוגות הבאים:

A, A

A, B

A, C

B, A

B, B

B, C

C, A

C, B

C, C

כמובן, זוג "עצמי" (X, X) אשר בו מופיע אותו קוד עובד בשני הטורים, אינו זוג של שני עובדים שונים. כמו כן, לכל זוג (X, Y), תמצא את זוג ה"ראי" שלו (Y, X) בתוצאה. עליך להחזיר רק אחד מהשניים. כדי לטפל בשתי הסוגיות, תוכל להגדיר תנאי join המסנן זוגות בהם המפתח מהטבלה השמאלית קטן מהמפתח מהטבלה הימנית. זוגות בהם אותו עובד מופיע פעמיים יוסרו. כמו כן, יוסר אחד מזוגות הראי (X, Y) ו–(Y, X) מכיוון שרק לאחד מהם יהיה מפתח שמאלי קטן מהמפתח הימני.

השאילתה הבאה מחזירה את התוצאה הרצויה, ללא זוגות ראי וללא זוגות עצמיים:

```
SELECT E1.EmployeeID, E1.LastName, E1.FirstName,
  E2.EmployeeID, E2.LastName, E2.FirstName
FROM dbo.Employees AS E1
  JOIN dbo.Employees AS E2
    ON E1.EmployeeID < E2.EmployeeID;
```

תוכל גם לחשב מספרי שורה על ידי שימוש ב–nonequijoin. כדוגמה, השאילתה הבאה מחשבת מספרי שורה עבור הזמנות מטבלת Orders, בהתבסס על OrderID גדל:

```
SELECT O1.OrderID, O1.CustomerID, O1.EmployeeID, COUNT(*) AS rn
FROM dbo.Orders AS O1
  JOIN dbo.Orders AS O2
    ON O2.OrderID <= O1.OrderID
GROUP BY O1.OrderID, O1.CustomerID, O1.EmployeeID;
```

תוכל למצוא קווי דמיון בין פתרון זה לבין הפתרון מבוסס–הסטים טרום SQL Server 2005 המשתמש בתת–שאילתות שהצגתי בפרק הקודם. לאחר שהופעלו שני השלבים הראשונים בעיבוד הלוגי של שאילתה (מכפלה קרטזית ומסנן ON), כל הזמנה מ–O1 מחוברת לכל ההזמנות מ–O2 להן OrderID שווה או קטן יותר. המשמעות היא ששורה מ–O1 עם מספר

שורת יעד n תחובר ל–n שורות מ–O2. כל שורה מ–O1 תשוכפל בתוצאה של ה–join n פעמים. אם זה נשמע מבלבל, עקוב אחרי כשאני מדגים לוגיקה זו בעזרת דוגמה. נאמר שיש לך הזמנות עם הקודים הבאים (לפי הסדר, מימין לשמאל): x, y ו–z. תוצאת ה–join תהיה כלהלן:

x, x

y, x

y, y

z, x

z, y

z, z

ה–join ייצר שכפולים לכל שורה מ–O1 – כמספר השורה המיועדת. השלב הבא הוא להביא לקריסה של כל קבוצת שורות חזרה לשורה אחת, תוך החזרת ספירה של השורות כמספר השורה:

x, 1

y, 2

z, 3

שים לב שעליך לכלול בפסוקית GROUP BY את כל המאפיינים מ–O1 שברצונך להחזיר. זכור שבשאילתת צבירה, מאפיין שברצונך להחזיר ברשימת ה–SELECT חייב להופיע בפסוקית GROUP BY. שאילתה זו סובלת מאותה סוגיית ביצועים של N^2 אותה תיארתי בפתרון תת–השאילתה. שאילתה זו גם מדגימה טכניקת "פרישה–קריסה" (expand-collapse), בה ה–join משיג את הפרישה של מספר השורות על ידי יצירת שכפולים, והקיבוץ משיג את קריסת השורות המאפשרת לך לחשב צבירות.

אני מעדיף את שיטת תת–השאילתות מכיוון שהיא הרבה יותר אינטואיטיבית. אני מוצא את שיטת ה"פרישה–קריסה" מלאכותית ובלתי אינטואיטיבית.

זכור שבשני הפתרונות ליצירת מספרי שורה השתמשת בפונקציית צבירה – ספירה של שורות. לוגיקה דומה מאוד יכולה לשמש לחישוב צבירות אחרות, או בעזרת תת–שאילתה או בעזרת join (שיטת פרישה–קריסה). ארחיב בנוגע לשיטה זו בפרק 6 בסעיף "צבירות נעות". אתאר שם גם תרחישים בהם עדיין אשקול שימוש בשיטת "פרישה–קריסה" למרות שאני מוצא אותה פחות אינטואיטיבית מאשר שיטת תת–השאילתה.

Joins מרובים

שאילתה עם מספר joins מערבת שלוש טבלאות או יותר. בסעיף זה, אתאר הן את ההיבטים הפיסיים והן את ההיבטים הלוגיים של שאילתות מרובות–joins.

שליטה בסדר הערכה (evaluation order) פיסי של Joins

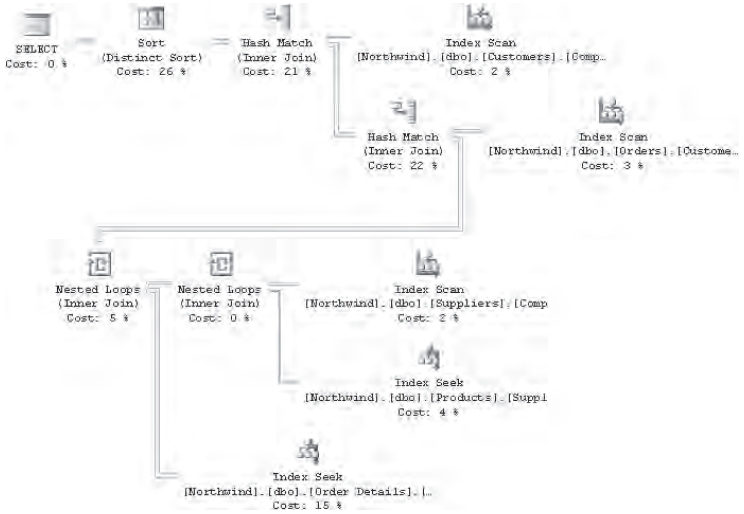
בשאילתה מרובת–joins ללא outer joins, אתה יכול לארגן מחדש את הסדר שבו מוגדרות הטבלאות ללא השפעה על התוצאה. ה–optimizer מודע לכך ויקבע את הסדר שבו הוא ניגש לטבלאות בהתבסס על הערכות עלות. בתוכנית העבודה של השאילתה, אתה עשוי למצוא שה–optimizer בחר לגשת לטבלאות בסדר שונה מזה שהגדרת בשאילתה.

למשל, השאילתה הבאה מחזירה שם חברה של לקוח ושם חברה של ספק, כאשר הספק סיפק מוצרים ללקוח:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
  JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID;
```

בחן את תוכנית העבודה המוצגת בתרשים 3–5, ותמצא שהגישה לטבלאות מתבצעת פיסית בסדר שונה מאשר הסדר הלוגי המוגדר בשאילתה.

תרשים 3–5: תוכנית עבודה לשאילתה מרובת–joins

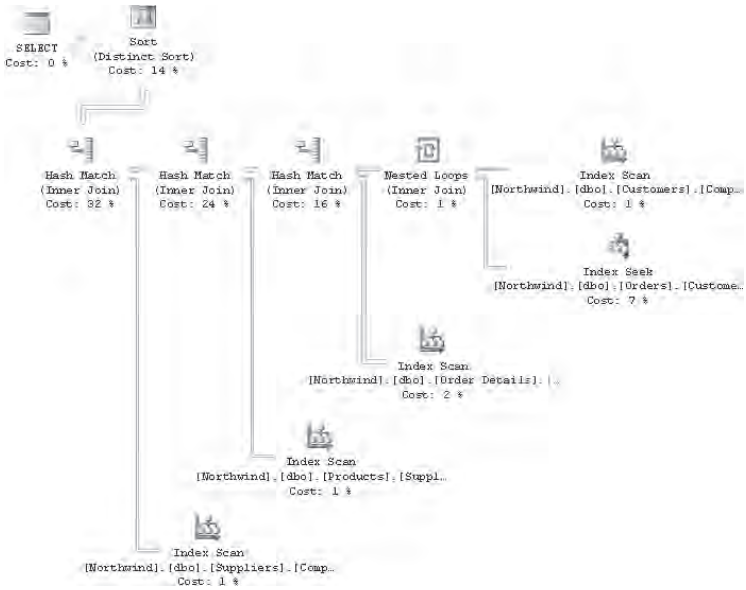


אם אתה חושד שתוכנית הניגשת לטבלאות בסדר שונה מזה שנבחר על ידי ה–optimizer תהיה יעילה יותר, תוכל לכפות את הסדר של עיבוד ה–joins על ידי שימוש באחת משתי אפשרויות. תוכל להשתמש ב– FORCE ORDER hint כלהלן, דבר שיכפה על ה–optimizer לעבד את ה–joins פיסית באותו סדר כמו זה הלוגי:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
  JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID
OPTION (FORCE ORDER);
```

שאילתה זו מייצרת את תוכנית העבודה המוצגת בתרשים 4–5, בו תוכל לראות שהגישה לטבלאות נעשית בסדר בו הן מופיעות בשאילתה:

תרשים 4–5: תוכנית עבודה לשאילתה מרובת–joins, כפיית סדר



אפשרות נוספת לכפות סדר של עיבוד joins היא להדליק את אפשרות ה– SET FORCEPLAN session. דבר זה ישפיע על כל השאילתות ב–session.

Hints

שים לב שבאופן כללי, שימוש ב–hints לדריסת הבחירה של ה–optimizer בתוכנית צריכה להיות אמצעי אחרון כאשר מתמודדים עם סוגיות ביצועים. hint אינו מחווה של רצון טוב; אלא, אתה מכריח את ה–optimizer להשתמש במסלול מסוים באופטימיזציה. אם אתה מכניס hint לקוד התפעולי שלך, אספקט זה של התוכנית הופך להיות סטטי (למשל, שימוש באינדקס מסוים או באלגוריתם join כלשהו). ה–optimizer לא יבצע בחירות דינמיות שיאפשרו שינויים בנפח ובפיזור נתונים.

קיימות מספר סיבות בגללן ה–optimizer עשוי לא לייצר תוכנית אופטימלית וש–hint עשוי לשפר ביצועים.

ראשית, ה–optimizer לא בהכרח מייצר את כל תוכניות העבודה האופטימליות לשאילתה. אם היה עושה כך, שלב האופטימיזציה פשוט היה עשוי לארוך זמן רב מדי. ה–optimizer מחשב סף לאופטימיזציה בהתבסס על גודל טבלאות הקלט, והוא יפסיק לבצע אופטימיזציה כאשר יגיע לסף זה, תוך הפקת התוכנית בעלת העלות הנמוכה ביותר מבין אלו שיצר. המשמעות היא שלא בהכרח תקבל את התוכנית האופטימלית.

שנית, במקרים רבים אופטימיזציה מבוססת על מידע סלקטיביות וצפיפות של נתונים, במיוחד בהקשר לבחירה באינדקסים ובשיטות גישה. אם סטטיסטיקות אינן מעודכנות או שאין להן אחוז דגימה מספק, ה–optimizer עלול לבצע השערות לא מדויקות.

שלישית, להיסטוגרמות הפיזור העיקריות ש– SQL Server מחזיק לטורי אינדקס (ובמקרים מסוימים גם לכאלו שאינם באינדקס) יש 200 מדרגות לכל היותר. עם תנאי join ומסננים רבים, ההבדל בין מידע הסלקטיביות/צפיפות שה–optimizer מעריך לבין המידע בפועל עשוי להיות משמעותי במקרים מסוימים, דבר המוביל לתוכניות בלתי יעילות.

זכור עם זאת, שלא מובטח שתקבל את התוכנית האופטימלית. עליך לעשות כל שבאפשרותך להימנע משימוש ב–hints בקוד תפעולי – למשל, וידוא שהסטטיסטיקות מעודכנות, הגדלת אחוז הדגימה אם צריך, ובמקרים מסוימים שינוי השאילתה כדי לעזור ל–optimizer לעשות בחירות טובות יותר. השתמש ב–hint רק כמוצא אחרון אם כל האמצעים האחרים נכשלו. ואם בסופו של דבר אתה אכן משתמש ב–hint, חזור ובחן את הקוד מפעם לפעם לאחר ביצוע חקירה נוספת או לאחר פתיחת קריאת תמיכה במיקרוסופט.

שליטה בסדר הערכה (evaluation order) לוגי של Joins

קיימים גם מקרים בהם ייתכן שתרצה לשלוט בסדר הלוגי של עיבוד joins מעבר לסדר בו מצוינות הטבלאות בפסוקית FROM. למשל, חשוב על הבקשה הקודמת להחזיר את כל הזוגות של שם חברה של הלקוח ושם חברה של הספק, כאשר הספק סיפק מוצרים ללקוח. נניח שהיית מתבקש גם להחזיר לקוחות שלא ביצעו הזמנות כלל. באינטואיציה, סביר להניח שהיית עושה את הניסיון הבא, המשתמש ב– left outer join בין Customers ל–Orders:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
  JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID;
```

השאילתה הקודמת החזירה 1,236 זוגות של לקוח–ספק, וציפית שזו תחזיר 1,238 שורות (מכיוון שישנם שני לקוחות שלא ביצעו הזמנות כלל). אף על פי כן, שאילתה זו מחזירה את אותו סט תוצאה כמו הקודמת ללא הלקוחות החיצוניים. זכור שה–join הראשון מתרחש רק בין שתי הטבלאות הראשונות (Customers ו–Orders), כששלושת שלבי העיבוד הלוגי של שאילתה מופעלים, ומופקת טבלה וירטואלית, אז מבוצע join עם הטבלה השלישית ([Order Details]) על טבלת התוצאה הווירטואלית, וכך הלאה.

ה–join הראשון אכן ייצר שורות חיצוניות ללקוחות ללא הזמנות, אך ה–OrderID בשורות חיצוניות אלו היה NULL, כמובן. ה–join השני – בין טבלת התוצאה הווירטואלית לבין [Order Details] – הסיר את השורות החיצוניות הללו מכיוון ש–equijoin לעולם לא ימצא התאמה המבוססת על השוואה ל–NULL.

קיימות מספר דרכים לוודא שלקוחות חיצוניים אלו לא ייעלמו. שיטה אחת היא להשתמש ב– left outer join בכל ה–joins, למרות שלוגית אתה רוצה inner joins בין Orders, [Order Details], Products ו–Suppliers:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
  LEFT OUTER JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
  LEFT OUTER JOIN dbo.Products AS P
```

```
    ON P.ProductID = OD.ProductID
  LEFT OUTER JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID;
```

ה– left outer join ישמור את הלקוחות בטבלאות הביניים הווירטואליות. שאילתה זו מייצרת נכונה 1,238 שורות, כולל שני הלקוחות שלא ביצעו הזמנות. עם זאת, אם היו לך הזמנות ללא פרטי הזמנה מקושרים, פרטי הזמנה ללא מוצרים מקושרים, או מוצרים ללא ספקים מקושרים, שאילתה זו הייתה מייצרת תוצאות שגויות. כלומר, היית מקבל שורות חיצוניות שלא רצית לקבל.

אפשרות נוספת היא לוודא שה–join עם טבלת Customers מעובד לוגית אחרון. ניתן להשיג זאת על ידי שימוש ב– inner joins בין כל הטבלאות, ולבסוף ב– right outer join עם Customers:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
  JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID
  RIGHT OUTER JOIN dbo.Customers AS C
    ON O.CustomerID = C.CustomerID;
```

תרחיש זה היה פשוט למדי, אך במקרים בהם אתה מערבב סוגים שונים של joins – שלא לדבר על אופרטורי טבלאות חדשים שנוספו ב– SQL Server 2005 (למשל, APPLY, PIVOT, UNPIVOT) – הדברים עשויים להיות לא כל–כך פשוטים. יתרה מכך, שימוש ב– left outer join לאורך כל הדרך הוא מלאכותי מאוד. אינטואיטיבי יותר לחשוב על השאילתה כ– left outer join יחיד. כאשר הטבלה השמאלית היא טבלת Customers והטבלה הימנית היא התוצאה של inner join בין כל הטבלאות האחרות. הן ANSI SQL והן T-SQL מאפשרים לך לשלוט בסדר הלוגי של עיבוד joins:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN
    (dbo.Orders AS O
       JOIN dbo.[Order Details] AS OD
         ON OD.OrderID = O.OrderID
       JOIN dbo.Products AS P
         ON P.ProductID = OD.ProductID
```

```
 JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID)
ON O.CustomerID = C.CustomerID;
```

טכנית, קיימת כאן התעלמות מהסוגריים, אך אני ממליץ לך להשתמש בהם מכיוון שהם יסייעו לך לכתוב את השאילתה בצורה נכונה. השימוש בסוגריים גרם לך לשנות גם היבט אחר של השאילתה, שהוא זה בו השפה באמת משתמשת לקבוע את סדר העיבוד הלוגי. אם לא ניחשת עדיין, זהו הסדר של הפסוקית ON. הגדרת הפסוקית ON – ON O.CustomerID = C.CustomerID – אחרונה גורמת ל– outer joins להיות מעובדים לוגית ראשונים; ה– left outer join מתרחש לוגית בין Customers לבין תוצאת ה– inner join בין יתר הטבלאות. באפשרותך לכתוב את השאילתה ללא סוגריים, ותהיה לה אותה משמעות:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN
    dbo.Orders AS O
       JOIN dbo.[Order Details] AS OD
         ON OD.OrderID = O.OrderID
       JOIN dbo.Products AS P
         ON P.ProductID = OD.ProductID
       JOIN dbo.Suppliers AS S
         ON S.SupplierID = P.SupplierID
      ON O.CustomerID = C.CustomerID;
```

וריאציות אחרות המגדירות את הפסוקית ON המתייחסת ל–C.CustomerID אחרון, כוללות את השתיים הבאות:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
  JOIN dbo.Products AS P
  JOIN dbo.[Order Details] AS OD
    ON P.ProductID = OD.ProductID
    ON OD.OrderID = O.OrderID
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID
    ON O.CustomerID = C.CustomerID;
```

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
  JOIN dbo.Products AS P
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID
    ON P.ProductID = OD.ProductID
    ON OD.OrderID = O.OrderID
    ON O.CustomerID = C.CustomerID;
```

ישנו חיסרון ברור באי שימוש בסוגריים – פגיעה בקריאות ובבהירות של הקוד. ללא סוגריים, השאילתות רחוקות מלהיות אינטואיטיביות. אך ישנה גם סוגיה אחרת.

שים לב: אינך יכול לשחק עם הסדר של הפסוקית ON כפי שתרצה. קיים יחס מסוים שחייב להישמר בין הסדר של הטבלאות המצוינות לבין הסדר של פסוקיות ה–ON המצוינת, כדי שהשאילתה תהיה תקינה. היחס נקרא **יחס כיאסטי**. יחס כיאסטי אינו ייחודי לא ל–T–SQL ולא למדעי המחשב; יחס זה מופיע בתחומים רבים, ביניהם שירה, לשון, מתמטיקה ואחרים. בסדרה ממוינת של פריטים, היחס מקשר את הפריט הראשון עם האחרון, השני עם זה שלפני אחרון, וכך הלאה. למשל, במילה ABBA מתקיים יחס כיאסטי בין האותיות. כדוגמה ליחס כיאסטי במתמטיקה, היזכר ברצף האריתמטי שתיארתי בפרק האחרון: 1,2,3,...,n. לחישוב סכום האיברים, אתה מרכיב n/2 זוגות בהתבסס על יחס כיאסטי (1 + n, 2+ n-1, 3 + n-2 וכך הלאה). הסכום של כל זוג הוא תמיד 1 + n; לפיכך, הסכום הכולל של הרצף האריתמטי הוא $(1 + n) * n / 2 = (n + n^2)/ 2$.

בדומה, היחס בין הטבלאות המצוינות בפסוקית FROM לבין פסוקיות ה–ON חייב להיות כיאסטי כדי שהשאילתה תהיה תקינה. כלומר, פסוקית ה–ON הראשונה יכולה להתייחס רק לשתי טבלאות הביניים שישר מעליה. פסוקית ה–ON השנייה יכולה להתייחס לשתי הטבלאות הקודמות ולאחת נוספת מייד מעליהן, וכך הלאה. תרשים 5–5 מציג את היחס הכיאסטי בשאילתה האחרונה. הקוד בתרשים אורגן מעט מחדש לצורך קריאות.

תרשים 5–5: יחס כיאסטי בשאילתה מרובת–joins

```
SELECT DISTINCT
   C.CompanyName AS customer,
   S.CompanyName AS supplier
 FROM
   dbo.Customers            AS  C   LEFT OUTER   JOIN
   dbo.Orders               AS  O                JOIN
   dbo.[Order Details]      AS  OD               JOIN
   dbo.Products AS P JOIN
   dbo.Suppliers AS S
     ON  S.SupplierID   =   P.SupplierID
     ON  P.ProductID    =  OD.ProductID
     ON OD.OrderID      =   O.OrderID
     ON  O.CustomerID   =   C.CustomerID;
```

ללא סוגריים, השאילתות אינן קריאות כל–כך ועליך להיות מודע ליחס הכיאסטי כדי לכתוב שאילתה תקינה. ולהיפך, אם אתה אכן משתמש בסוגריים, השאילתות קריאות ואינטואיטיביות יותר ואינך צריך לדאוג ליחס הכיאסטי, שכן הסוגריים כופות עליך לכתוב נכון.

Semi Join

semi joins הם joins המחזירים שורות מטבלה אחת בהתבסס על קיומן של שורות קשורות בטבלה האחרת. אם אתה מחזיר מאפיינים מהטבלה השמאלית, ה–join נקרא left semi join. אם אתה מחזיר מאפיינים מהטבלה הימנית, הוא נקרא right semi join.

קיימות מספר דרכים להשגת semi join: שימוש ב– inner joins, תת–שאילתות ופעולות–סטים (אותן אדגים בהמשך הפרק). בשימוש ב– inner join, אתה בוחר במאפיינים רק מאחת הטבלאות ומפעיל DISTINCT. למשל השאילתה הבאה מחזירה לקוחות מספרד שעשו הזמנות:

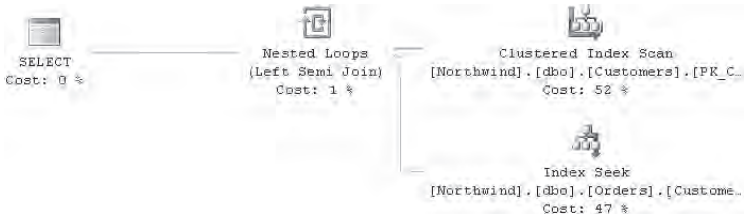
```
SELECT DISTINCT C.CustomerID, C.CompanyName
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
WHERE Country = N'Spain';
```

תוכל גם להשתמש בביטוי EXIST כלהלן:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND EXISTS
    (SELECT * FROM dbo.Orders AS O
     WHERE O.CustomerID = C.CustomerID);
```

אם אתה תוהה האם קיימים הבדלי ביצועים בין השתיים, במקרה זה ה–optimizer מייצר תוכנית זהה לשתיהן. תוכנית זו מוצגת בתרשים 6–5.

תרשים 6–5: תוכנית עבודה עבור left semi join



ההופכי של semi join הוא anti-semi join, כאשר אתה מחפש אחר שורות בטבלה אחת בהתבסס על אי–קיומן באחרת. ניתן להשיג anti-semi join (left או right) על ידי שימוש ב– outer join, המסנן רק שורות חיצוניות. למשל, השאילתה הבאה מחזירה לקוחות מספרד שלא ביצעו הזמנות. ה– anti-semi join מושג על ידי שימוש ב– outer join:

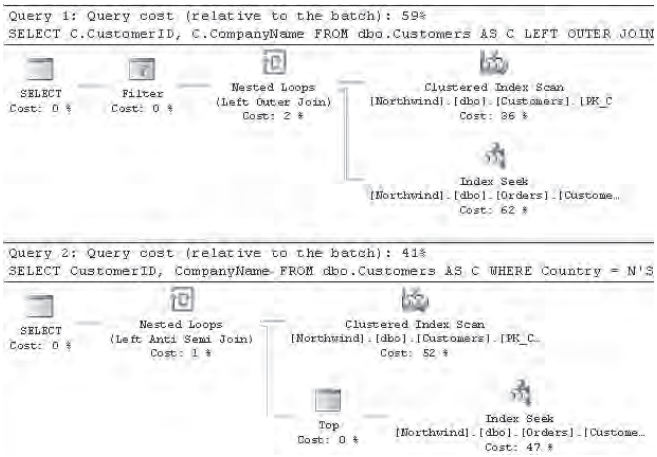
```
SELECT C.CustomerID, C.CompanyName
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
WHERE Country = N'Spain'
  AND O.CustomerID IS NULL;
```

ניתן גם להשתמש בביטוי NOT EXIST כלהלן:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND NOT EXISTS
    (SELECT * FROM dbo.Orders AS O
     WHERE O.CustomerID = C.CustomerID);
```

כפי שתוכל לראות בתוכניות העבודה המוצגות בתרשים 7–5 עבור שתי הווריאציות של השאילתה, לפתרון המשתמש בביטוי NOT EXIST ביצועים טובים יותר.

תרשים 7–5: תוכנית עבודה עבור left anti-semi join



התוכנית עבור פתרון ה– outer join מראה שכל ההזמנות של לקוחות מספרד עובדו בפועל. הנח ש–c שווה למספר הלקוחות מספרד, ו–o שווה למספר ההזמנות הממוצע ללקוח. תקבל שהייתה גישה ל– c x o הזמנות. אז מסוננות רק השורות החיצוניות.

התוכנית עבור פתרון ה– NOT EXIST יעילה יותר. כמו התוכנית עבור הפתרון LEFT OUTER JOIN, התוכנית מבצעת seek בתוך האינדקס על Orders.CustomerID לכל לקוח. עם זאת, התוכנית של NOT EXIST בודקת רק האם שורה עם קוד לקוח זה נמצאה או לא (מוצג על ידי האופרטור TOP) – בעוד שהתוכנית עבור ה– outer join סורקת בפועל את כל שורות האינדקס לכל לקוח.

כמות כוללת נעה של שנה קודמת

התרגיל הבא מדגים תערובת של מספר קטגוריות join: self join, nonequijoin ושאילתה מרובת–joins. ראשית צור את הטבלה MonthlyOrders ומלא אותה בנתונים על ידי הרצת הקוד בקטע–קוד 2–5.

קטע–קוד 2–5: יצירה ומילוי של טבלת MonthlyOrders

```
IF OBJECT_ID('dbo.MonthlyOrders') IS NOT NULL
  DROP TABLE dbo.MonthlyOrders;
GO

SELECT
  CAST(CONVERT(CHAR(6), OrderDate, 112) + '01' AS DATETIME)
    AS ordermonth,
  COUNT(*) AS numorders
```

```
INTO dbo.MonthlyOrders
FROM dbo.Orders
GROUP BY CAST(CONVERT(CHAR(6), OrderDate, 112) + '01' AS DATETIME);

CREATE UNIQUE CLUSTERED INDEX idx_ordermonth ON dbo.MonthlyOrders(ordermonth);
```

הטבלה מאחסנת את הכמות הכוללת של הזמנות לכל חודש כפי שמוצג בטבלה 5–5.

טבלה 5–5: תוכן טבלת MonthlyOrders

ordermonth	*numorders*
1996-07-01 00:00:00.000	22
1996-08-01 00:00:00.000	25
1996-09-01 00:00:00.000	23
1996-10-01 00:00:00.000	26
1996-11-01 00:00:00.000	25
1996-12-01 00:00:00.000	31
1997-01-01 00:00:00.000	33
1997-02-01 00:00:00.000	29
1997-03-01 00:00:00.000	30
1997-04-01 00:00:00.000	31
1997-05-01 00:00:00.000	32
1997-06-01 00:00:00.000	30
1997-07-01 00:00:00.000	33
1997-08-01 00:00:00.000	33
1997-09-01 00:00:00.000	37
1997-10-01 00:00:00.000	38
1997-11-01 00:00:00.000	34
1997-12-01 00:00:00.000	48
1998-01-01 00:00:00.000	55
1998-02-01 00:00:00.000	54
1998-03-01 00:00:00.000	73
1998-04-01 00:00:00.000	74
1998-05-01 00:00:00.000	14

שים לב שהשתמשתי בטיפוס הנתונים DATETIME עבור הטור ordermonth. תאריך תקין חייב לכלול חלק של יום. אחסנת חודש ההזמנה בטיפוס נתונים DATETIME מאפשר מניפולציות גמישות יותר בעזרת שימוש בפונקציות DATETIME.

הבקשה היא להחזיר, לכל חודש, כמות כוללת נעה של השנה הקודמת. במילים אחרות, לכל חודש n, החזר את כמות ההזמנות הכוללת מחודש n פחות 11 ועד חודש n.

הפתרון שאציג אינו השיטה היעילה ביותר להשגת משימה זו. אני מציג אותו כאן כדוגמה לשילוב בין סוגים וקטגוריות שונים של joins. כמו כן, אני מניח שלא קיימים פערים ברצף החודשים. בפרק 6 תמצא דיון ממוקד בנושא צבירות נעות, כולל סוגיות ביצועים.

השאילתה הבאה מחזירה את הכמות הכוללת הנעה של השנה הקודמת לכל חודש, ומייצרת את הפלט המופיע בטבלה 6–5:

```
SELECT
  CONVERT(CHAR(6), O1.ordermonth, 112) AS frommonth,
  CONVERT(CHAR(6), O2.ordermonth, 112) AS tomonth,
  SUM(O3.numorders) AS numorders
FROM dbo.MonthlyOrders AS O1
  JOIN dbo.MonthlyOrders AS O2
    ON DATEADD(month, -11, O2.ordermonth) = O1.ordermonth
  JOIN dbo.MonthlyOrders AS O3
    ON O3.ordermonth BETWEEN O1.ordermonth AND O2.ordermonth
GROUP BY O1.ordermonth, O2.ordermonth;
```

טבלה 6–5: כמות כוללת נעה של שנה קודמת

frommonth	*tomonth*	*numorders*
199607	199706	337
199608	199707	348
199609	199708	356
199610	199709	370
199611	199710	382
199612	199711	391
199701	199712	408
199702	199801	430
199703	199802	455
199704	199803	498
199705	199804	541
199706	199805	523

השאילתה הראשית מחברת שני מופעים של MonthlyOrders, O1 ו–O2. שני המופעים מספקים תאריכי גבול של השנה הנעה, O1 עבור הגבולות התחתונים (frommonth), ו–O2 עבור הגבולות העליונים (tomonth). לפיכך, תנאי ה–join הוא כלהלן: ordermonth in O1 = ordermonth in O2-11 months. למשל, יולי 1996 ב–O1 יקושר עם יוני 1997 ב–O2.

לאחר שהגבולות נקבעו, join אחר, למופע שלישי של MonthlyOrders (O3), מקשר לכל זוג–גבולות שורות הנופלות בתחום זה. במילים אחרות, כל זוג–גבולות ימצא 12 התאמות, אחת לכל חודש, בהנחה שהיו הזמנות בכל 12 החודשים. הלוגיקה כאן דומה לשיטת הפרישה עליה דיברתי קודם. כעת לאחר שכל זוג–גבולות שוכפל 12 פעמים, פעם אחת לכל חודש מ–O3, תרצה שהקבוצה תקרוס חזרה לשורה יחידה, המחזירה את מספר ההזמנות הכולל לכל קבוצה.

שים לב שפתרון זה יחזיר רק זוגות בהם שני הגבולות קיימים בנתונים וביניהם 11 חודשים. הוא לא יחזיר חודשי גבול–גבוה עבורם לא קיים בנתונים חודש גבול–נמוך. יולי 1996 הוא כרגע החודש המוקדם ביותר הקיים בטבלה. לפיכך, יוני 1997 הוא חודש הגבול–הגבוה הראשון המופיע בסט התוצאה. אם ברצונך לקבל תוצאות גם עם tomonth לפני יוני 1997, עליך לשנות את ה–join הראשון ל– right outer join. ה– right outer join יציב NULL בטור frommonth עבור השורות החיצוניות. כדי לא לאבד שורות חיצוניות אלו ב–join השני, בתנאי ה–join אתה ממיר NULL בטור frommonth לתאריך 1 בינואר 1900, ובכך אתה מבטיח ש– frommonth <= tomonth. אפשרות נוספת תהיה להשתמש בחודש הקטן ביותר בתחום של 12 החודשים שעליהם מחושבת הכמות הכוללת. לצורך הפשטות, אשתמש באפשרות הראשונה.

בדומה, ברשימת ה–SELECT אתה ממיר NULL בטור frommonth לחודש הקטן ביותר שקיים. כדי לציין שתחומים מסוימים לא מכסים שנה שלמה, החזר גם את מספר החודשים המעורבים בצבירה. להלן הפתרון המלא, המחזיר את הפלט המוצג בטבלה 7–5:

```
SELECT
  CONVERT(VARCHAR(6),
    COALESCE(O1.ordermonth,
      (SELECT MIN(ordermonth) FROM dbo.MonthlyOrders)),
    112) AS frommonth,
  CONVERT(VARCHAR(6), O2.ordermonth, 112) AS tomonth,
  SUM(O3.numorders) AS numorders,
  DATEDIFF(month,
    COALESCE(O1.ordermonth,
      (SELECT MIN(ordermonth) FROM dbo.MonthlyOrders)),
    O2.ordermonth) + 1 AS nummonths
FROM dbo.MonthlyOrders AS O1
  RIGHT JOIN dbo.MonthlyOrders AS O2
    ON DATEADD(month, -11, O2.ordermonth) = O1.ordermonth
```

```
  JOIN dbo.MonthlyOrders AS O3
    ON O3.ordermonth BETWEEN COALESCE(O1.ordermonth, '19000101')
                         AND O2.ordermonth
GROUP BY O1.ordermonth, O2.ordermonth;
```

טבלה 7–5: כמות כוללת נעה של שנה קודמת, כולל כל החודשים

frommonth	*tomonth*	*numorders*	*nummonths*
199607	199607	22	1
199607	199608	47	2
199607	199609	70	3
199607	199610	96	4
199607	199611	121	5
199607	199612	152	6
199607	199701	185	7
199607	199702	214	8
199607	199703	244	9
199607	199704	275	10
199607	199705	307	11
199607	199706	337	12
199608	199707	348	12
199609	199708	356	12
199610	199709	370	12
199611	199710	382	12
199612	199711	391	12
199701	199712	408	12
199702	199801	430	12
199703	199802	455	12
199704	199803	498	12
199705	199804	541	12
199706	199805	523	12

לניקוי, הסר את טבלת MonthlyOrders:

```
DROP TABLE dbo.MonthlyOrders;
```

אלגוריתמים של Join

אלגוריתמים של join הם האסטרטגיות הפיסיות בהן יכול SQL Server להשתמש לעיבוד joins. לפני SQL Server 7.0, נתמך רק אלגוריתם join אחד (הנקרא nested loops או loop join). מאז גרסה 7.0, SQL Server תומך גם באלגוריתמים merge ו– hash join.

Loop Join

loop join סורק את אחת מהטבלאות המוצלבות (הטבלה העליונה בתוכנית העבודה הגרפית), ולכל שורה הוא מחפש את השורות התואמות בטבלה המוצלבת האחרת (הטבלה התחתונה בתוכנית).

שים לב: הנוכחות של אופרטור loop join בתוכנית העבודה אינה מהווה אינדיקציה לשאלה האם התוכנית יעילה או לא. loop join הוא אלגוריתם ברירת מחדל שיכול תמיד להיות מיושם. לאלגוריתמים האחרים קיימות דרישות – למשל, ה–join חייב להיות equijoin.

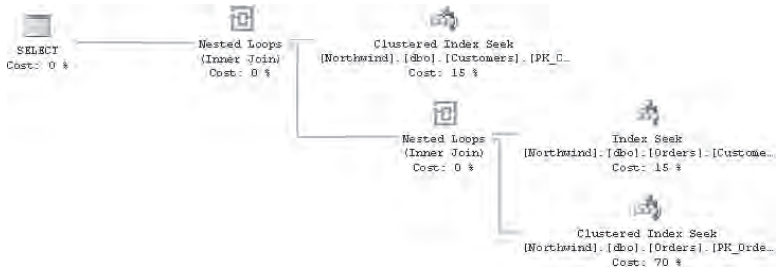
שימוש ב– loop join יעיל כאשר יש אינדקס על טור ה–join בטבלה הגדולה יותר, דבר שמאפשר seek שלאחריו סריקה חלקית. מידת היעילות תלויה במיקום האינדקס בסרגל האופטימיזציה של האינדקס:

ביצועים גרועים ביותר ← אין אינדקס (סריקת טבלה לכל שורה חיצונית) ←
← nonclustered noncovering (seek + partial ordered scan + lookups)
← clustered (seek + partial ordered scan)
← nonclustered covering (seek + partial ordered scan)
← nonclustered covering with included nonkey columns (seek + partial ordered scan)
ביצועים טובים ביותר.

השאילתה הבאה, המפיקה את התוכנית המוצגת בתרשים 8–5, היא דוגמה לשאילתה עבורה ה–optimizer בוחר באופרטור loop join:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID, O.OrderDate
FROM dbo.Customers AS C
 JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
WHERE C.CustomerID = N'ALFKI';
```

תרשים 8–5: תוכנית עבודה עבור loop join



התוכנית מבצעת seek בתוך האינדקס–clustered על Customers כדי להחזיר את הלקוחות המסוננים. לכל לקוח (במקרה שלנו, קיים רק אחד), אופרטור loop join מאתחל seek בתוך האינדקס על Orders.CustomerID. לאחר ה–seek מופיעה סריקה חלקית להשגת מאתרי שורה (row locators) לכל הזמנות הלקוחות. לכל מאתר שורה, אופרטור loop join נוסף (מתחת ומימין לראשון) מאתחל lookup של שורת הנתונים. שים לב שאופרטור loop join זה אינו קשור לוגית ל–join בשאילתה; אלא, הוא משמש לצורך אתחול פעולות ה–lookup הפיסיות. נסה לחשוב על כך כעל join פנימי בתוך האינדקס–clustered על Orders, מכיוון שהטבלה היא clustered ומאתר השורה הוא מפתח ה–clustering.

שים לב: בהקשר ל–join ולאינדקסים, זכור ש–joins מבוססים לעיתים קרובות על יחסי מפתח זר/מפתח ראשי. בעוד שאינדקס (כדי לכפות ייחודיות) נוצר אוטומטית כאשר מפתח ראשי מוכרז, הרי שהכרזה על מפתח זר אינה יוצרת אינדקס אוטומטית. זכור שעבור loop joins, לרוב רצוי אינדקס על טור ה–join בטבלה הגדולה. כך שבאחריותך ליצור את האינדקס הזה בעצמך.

Merge Join

merge join הוא אלגוריתם join יעיל ביותר, הנשען על שני קלטים הממוינים על בסיס טורי ה–join. עם join אחד–לרבים, אופרטור merge join סורק כל קלט פעם אחת בלבד – ומכאן העליונות על פני אופרטורים אחרים. כדי לקבל שני קלטים ממוינים, ה–optimizer יכול להשתמש באינדקסים–clustered, או אפילו טוב יותר מכך, באינדקסים–מכסים–nonclustered, כאשר הטורים הראשונים הם טורי ה– join. SQL Server יתחיל לסרוק את שני הצדדים וינוע קדימה ראשית דרך צד ה"רבים" (נאמר, צד "אחד" והוא x וצד "רבים" הוא – משמאל לימין – x, x, x, x, y). SQL Server יעבור דרך צד "אחד" כאשר ערך טור ה–join על צד "רבים" משתנה (צד "אחד" עובר דרך y, ואז צד "רבים" עובר דרך – משמאל לימין – y, y, y, z). בסופו של דבר, כל צד נסרק רק פעם אחת, לפי הסדר. הדברים מסתבכים כאשר יש לך join רבים–לרבים, כאשר ה–optimizer עשוי עדיין להשתמש באופרטור merge join עם לוגיקה של rewind.

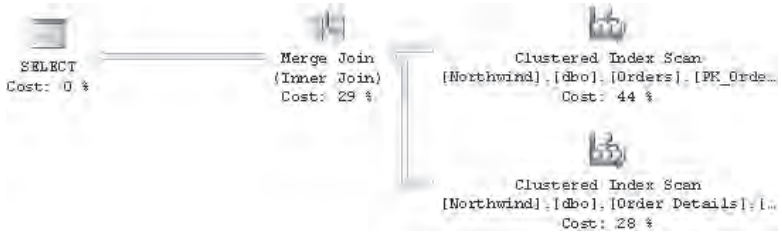
כאשר אתה רואה בתוכנית merge join, זהו לרוב סימן טוב.

כדוגמה, השאילתה הבאה מבצעת join בין Orders לבין [Order Details] על בסיס ערכי OrderID שווים:

```
SELECT O.OrderID, O.OrderDate, OD.ProductID, OD.Quantity
FROM dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
    ON O.OrderID = OD.OrderID;
```

לשתי הטבלאות קיים אינדקס–clustered על OrderID, כך שה–optimizer בוחר ב– merge join. האופרטור merge join מופיע בתוכנית עבור שאילתה זו, כפי שמוצגת בתרשים 9–5.

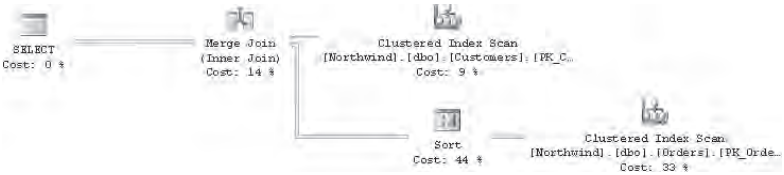
תרשים 9–5: תוכנית עבודה עבור merge join



במקרים מסוימים, ה–optimizer עשוי להחליט להשתמש ב– merge join אפילו כאשר אחד מהקלטים אינו ממוין מראש על ידי אינדקס, במיוחד אם קלט זה קטן למדי. במקרה כזה, תראה קלט זה נסרק ואז ממוין, כמו בתוכנית העבודה של השאילתה הבאה. תרשים 10–5 מציג את תוכנית העבודה לשאילתה זו:

```
SELECT C.CustomerID, CompanyName, ContactName, ContactTitle,
  OrderID, OrderDate
FROM dbo.Customers AS C
 JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID;
```

תרשים 10–5: תוכנית עבודה עבור merge join עם מיון



Hash Join

האופרטור hash join לרוב נבחר על ידי ה–optimizer כאשר חסרים אינדקסים טובים על טורי ה–join. אם אינך מייצר אינדקסים הולמים עבור ה–join, ה–optimizer מייצר טבלת hash כמבנה חיפוש אלטרנטיבי לעצים המאוזנים. עצים מאוזנים לרוב יעילים יותר כמבני חיפוש מאשר טבלאות hash, אך יקר יותר לייצר אותם. לעיתים, אתה רואה תוכניות עבודה בהן ה–optimizer מחליט שכדאי לייצר אינדקס זמני (אופרטור Index Spool). אך לרוב, עבור שאילתות אד–הוק, יקר יותר לייצר אינדקס זמני (עץ מאוזן), להשתמש בו ולהסיר אותו מאשר לייצר טבלת hash ולהשתמש בה.

ה–optimizer משתמש בקלט הקטן יותר מבין השניים כקלט הבנייה לטבלת ה–hash. הוא מפזר את השורות (טורים רלוונטיים לשאילתה) מקלט הבנייה לתוך buckets, בהתבסס על פונקציית hash המופעלת על ערכי טורי ה–join. פונקציית ה–hash נבחרת כדי ליצור מספר קבוע מראש של buckets בעלי גודל שווה פחות או יותר. נאמר שיש לך מחסן עם מספר גדול של כלים וחפצים. אם אינך מארגן אותם בצורה מסוימת, בכל פעם שאתה מחפש חפץ, עליך לסרוק את כולם. דבר זה דומה לסריקת טבלה. כמובן, תרצה לארגן את החפצים בקבוצות ובמדפים לפי קריטריון מסוים – למשל, לפי פונקציונליות, גודל, צבע, וכך הלאה. סביר להניח שתבחר בקריטריון שיביא לקבוצות בעלות גודל פחות או יותר שווה אשר קלות לטיפול.

הקריטריון בו תשתמש מקביל לפונקציית ה–hash, וקבוצה או מדף של חפצים מקביל ל–bucket. ברגע שהחפצים במחסן מאורגנים, בכל פעם שעליך לחפש אחד מהם, תפעיל עליו את אותו קריטריון שהפעלת לצורך ארגון החפצים, תיגש ישירות למדף הרלוונטי ותסרוק אותו.

טיפ: כאשר אתה רואה אופרטור hash join בתוכנית העבודה, הוא צריך לשמש לך כתמרור אזהרה שייתכן והנתונים שלך חסרים אינדקס חשוב. כמובן, אם אתה מריץ שאילתת אד–הוק שאינה חוזרת לעיתים קרובות, ייתכן שתסתפק ב– hash join. עם זאת, אם זוהי שאילתה תפעולית המופעלת לעיתים קרובות, ייתכן שתרצה ליצור את האינדקסים החסרים.

כדי להדגים hash join, ראשית צור העתקים של הטבלאות Orders ו– Order Details ללא אינדקסים על טורי ה–join:

```
SELECT * INTO dbo.MyOrders FROM dbo.Orders;
SELECT * INTO dbo.MyOrderDetails FROM dbo.[Order Details];
```

כעת, הרץ את השאילתה הבאה:

```
SELECT O.OrderID, O.OrderDate, OD.ProductID, OD.Quantity
FROM dbo.MyOrders AS O
  JOIN dbo.MyOrderDetails AS OD
    ON O.OrderID = OD.OrderID;
```

תוכל לראות את האופרטור Hash Match בתוכנית העבודה המיוצרת לשאילתה זו המוצגת בתרשים 11–5:

תרשים 11–5: תוכנית עבודה עבור hash join

כאשר תסיים, הסר את הטבלאות שזה עתה יצרת:

```
DROP TABLE dbo.MyOrders;
DROP TABLE dbo.MyOrderDetails;
```

אכיפת אסטרטגיית Join

באפשרותך לאלץ את ה–optimizer להשתמש באלגוריתם join מסוים, בתנאי שהוא נתמך טכנית עבור השאילתה הנתונה. אתה עושה זאת על ידי ציון hint בין מילת או מילות המפתח המייצגות את סוג ה–join (למשל, INNER, LEFT OUTER) לבין מילת המפתח JOIN. למשל, השאילתה הבאה מאלצת loop join:

```
SELECT C.CustomerID, C.CcompanyName, O.OrderID
FROM dbo.Customers AS C
 INNER LOOP JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID;
```

שים לב: עם inner joins, כאשר כופים אלגוריתם join, מילת המפתח INNER אינה אופציונלית. עם outer joins, מילת המפתח OUTER עדיין אופציונלית. למשל, תוכל לכתוב LEFT LOOP JOIN או LEFT OUTER LOOP JOIN.

בשימוש בתחביר join הישן, אינך יכול לציין אלגוריתם join שונה לכל join; אלא, אלגוריתם אחד ישמש לכל ה–joins בשאילתה. אתה עושה זאת על ידי שימוש בפסוקית OPTION, כלהלן:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE O.CustomerID = C.CustomerID
OPTION (LOOP JOIN);
```

טיפ: זכור את הדיון שערכנו מוקדם יותר בפרק בקשר לשימוש ב–hints כדי לדרוס את הבחירות של ה–optimizer. הגבל את השימוש בהם, ונסה למצות את כל האמצעים האחרים בטרם אתה מציג hint כזה בקוד תפעולי.

הפרדת אלמנטים (פירוק מערכים)

בנקודה זו, יש לך הזדמנות להשתמש בשיטות המפתח ובידע שצברת עד כה בנושא joins. אציג כאן צורה כללית של בעיה קשה למדי בעלת יישומים מעשיים רבים במערכות תפעוליות. צור ומלא בנתונים טבלה הנקראת Arrays על ידי הרצת הקוד בקטע–קוד 3–5.

קטע–קוד 3–5: יצירה ומילוי של הטבלה Arrays

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Arrays') IS NOT NULL
  DROP TABLE dbo.Arrays;
GO

CREATE TABLE dbo.Arrays
(
  arrid VARCHAR(10)    NOT NULL PRIMARY KEY,
  array VARCHAR(8000)  NOT NULL
)

INSERT INTO Arrays(arrid, array) VALUES('A', '20,22,25,25,14');
INSERT INTO Arrays(arrid, array) VALUES('B', '30,33,28');
INSERT INTO Arrays(arrid, array) VALUES('C', '12,10,8,12,12,13,12,14,10,9');
INSERT INTO Arrays(arrid, array) VALUES('D', '-4,-6,-4,-2');
```

הטבלה מכילה מערכים של אלמנטים מופרדים על ידי פסיקים. משימתך היא לכתוב שאילתה המייצרת את התוצאה המוצגת בטבלה 8–5.

טבלה 8–5: Arrays מפוצלים לאלמנטים

arrid	*pos*	*val*
A	1	20
A	2	22
A	3	25
A	4	25
A	5	14
B	1	30
B	2	33
B	3	28
C	1	12
C	2	10
C	3	8
C	4	12
C	5	12
C	6	13
C	7	12
C	8	14
C	9	10
D	1	-4
D	2	-6
D	3	-4
D	4	-2

הבקשה היא לנרמל נתונים שאינם ממלאים את חוק הנרמול הראשון – אסורים מאפיינים מרובי ערכים. סט התוצאה צריך להכיל שורה לכל אלמנט מערך, כולל קוד המערך, מיקום האלמנט במערך וערך האלמנט. הפתרון מוצג בפסקאות הבאות.

בטרם תתחיל לכתוב קוד, תמיד כדאי לזהות את הצעדים בפתרון ולפתור אותם לוגית. לעיתים קרובות נקודת פתיחה טובה היא לחשוב במונחים של מספר השורות ביעד, ולשקול מה היחס בין מספר זה למספר השורות במקור. מובן מאליו, כאן עליך לייצר מספר שורות בתוצאה מכל שורה במערך. במילים אחרות, כצעד ראשון, עליך לייצר שכפולים.

אתה כבר יודע שבשביל לייצר שכפולים, אתה יכול לאחד את טבלת Arrays עם טבלת עזר של מספרים. כאן ה–join אינו cross join פשוט ומסנן על מספר קבוע של שורות. מספר

השכפולים כאן צריך להיות שווה למספר האלמנטים במערך. כל אלמנט מזוהה על ידי פסיק שמקדים אותו (מלבד האלמנט הראשון, אותו אל לנו לשכוח). אם כך, תנאי ה–join יכול להתבסס על קיומו של פסיק במיקום התו ה–n במערך, כאשר n בא מטבלת Nums.

מובן מאליו שלא תרצה לבדוק תווים מעבר לאורך המערך, כך שאתה יכול להגביל את n כאורך המערך. השאילתה הבאה מיישמת את הצעד הראשון של הפתרון, ומפיקה את הפלט המוצג בטבלה 9–5:

```
SELECT arrid, array, n
FROM dbo.Arrays
  JOIN dbo.Nums
    ON n <= LEN(array)
    AND SUBSTRING(array, n, 1) = ',';
```

טבלה 9–5: פתרון לבעיית הפרדת אלמנטים, צעד 1

arrid	*array*	*n*
A	20,22,25,25,14	3
A	20,22,25,25,14	6
A	20,22,25,25,14	9
A	20,22,25,25,14	12
B	30,33,28	3
B	30,33,28	6
C	12,10,8,12,12,13,12,14,10,9	3
C	12,10,8,12,12,13,12,14,10,9	6
C	12,10,8,12,12,13,12,14,10,9	8
C	12,10,8,12,12,13,12,14,10,9	11
C	12,10,8,12,12,13,12,14,10,9	14
C	12,10,8,12,12,13,12,14,10,9	17
C	12,10,8,12,12,13,12,14,10,9	20
C	12,10,8,12,12,13,12,14,10,9	23
C	12,10,8,12,12,13,12,14,10,9	26
D	-4,-6,-4,-2	3
D	-4,-6,-4,-2	6
D	-4,-6,-4,-2	9

כמעט יצרת את מספר השכפולים הנכון לכל מערך, בצירוף ערך ה–n המייצג את מיקום הפסיק התואם. יש לך שכפול אחד פחות מאשר מספר השכפולים הרצוי לכל מערך. למשל, למערך A יש חמישה אלמנטים אך לך יש רק ארבע שורות. הסיבה שחסרה שורה לכל מערך היא שלא קיים פסיק מקדים לאלמנט הראשון במערך. כדי לתקן את הבעיה הקטנה הזו, שרשר פסיק ואת המערך ליצירת הקלט הראשון של הפונקציה SUBSTRING:

```
SELECT arrid, array, n
FROM dbo.Arrays
  JOIN dbo.Nums
    ON n <= LEN(array)
    AND SUBSTRING(',' + array, n, 1) = ',';
```

כפי שתוכל לראות בפלט המוצג בטבלה 10–5, כל מערך כעת מייצר שורה נוספת בתוצאה עם n = 1.

טבלה 10–5: פתרון לבעיית הפרדת אלמנטים, צעד 2

arrid	*array*	*N*
A	20,22,25,25,14	1
A	20,22,25,25,14	4
A	20,22,25,25,14	7
A	20,22,25,25,14	10
A	20,22,25,25,14	13
B	30,33,28	1
B	30,33,28	4
B	30,33,28	7
C	12,10,8,12,12,13,12,14,10,9	1
C	12,10,8,12,12,13,12,14,10,9	4
C	12,10,8,12,12,13,12,14,10,9	7
C	12,10,8,12,12,13,12,14,10,9	9
C	12,10,8,12,12,13,12,14,10,9	12
C	12,10,8,12,12,13,12,14,10,9	15
C	12,10,8,12,12,13,12,14,10,9	18
C	12,10,8,12,12,13,12,14,10,9	21
C	12,10,8,12,12,13,12,14,10,9	24

arrid	array	N
C	12,10,8,12,12,13,12,14,10,9	27
D	-4,-6,-4,-2	1
D	-4,-6,-4,-2	4
D	-4,-6,-4,-2	7
D	-4,-6,-4,-2	10

כמו כן, מכיוון שכל התווים ב– array + ',' ממוקמים תו אחד ימינה למיקומם במערך המקורי, כל ערכי n גדולים מערכם הקודם באחד. דבר זה אפילו טוב יותר עבורנו מכיוון שכעת n מייצג את המיקום ההתחלתי של האלמנט התואם בתוך המערך.

הצעד השלישי הוא לחלץ מכל שורה את האלמנט המתחיל בתו n. אתה יודע היכן האלמנט מתחיל – בתו ה–n – אך עליך לגלות את אורכו. אורך האלמנט הוא המיקום של הפסיק הבא פחות המיקום ההתחלתי של האלמנט (n). אתה משתמש בפונקציה CHARINDEX כדי למצוא את מיקום הפסיק הבא. תצטרך לספק לפונקציה את הערך n כארגומנט השלישי כדי לומר לה להתחיל לחפש את הפסיק בתו ה–n או אחריו, ולא מתחילת המחרוזת. זכור רק שתתקל כאן בבעיה דומה מאוד לזו שגרמה לך לקבל שכפול אחד פחות מאשר מספר האלמנטים. כאן, אין פסיק לאחר האלמנט האחרון. כפי שהוספת פסיק לפני האלמנט הראשון קודם לכן, תוכל להוסיף כעת פסיק אחד בסוף. השאילתה הבאה מציגה את הצעד השלישי בפתרון ומפיקה את הפלט המוצג בטבלה 11–5:

```
SELECT arrid,
  SUBSTRING(array, n, CHARINDEX(',', array + ',', n) - n) AS element
FROM dbo.Arrays
  JOIN dbo.Nums
    ON n <= LEN(array)
    AND SUBSTRING(',' + array, n, 1) = ',';
```

טבלה 11–5: פתרון לבעיית הפרדת אלמנטים, צעד 3

arrid	element
A	20
A	22
A	25
A	25
A	14
B	30

arrid	*element*
B	33
B	28
C	12
C	10
C	8
C	12
C	12
C	13
C	12
C	14
C	10
C	9
D	-4
D	-6
D	-4
D	-2

שים לב שטור התוצאה element הוא כרגע מחרוזת תווים. ייתכן שתרצה להמיר אותו לטיפוס נתונים הולם יותר (למשל, מספר שלם במקרה זה).

לבסוף, הצעד האחרון בפתרון הוא לחשב את המיקום של כל אלמנט בתוך המערך. זהו צעד ערמומי.

ראשית עליך לגלות מה קובע את מיקומו של אלמנט בתוך מערך. המיקום הוא מספר הפסיקים במערך המקורי עד לתו ה–n, ועוד אחד. ברגע שגילית זאת, עליך למצוא ביטוי שייחשב זאת. תרצה להימנע מלהשתמש בפונקציית T-SQL מוגדרת–משתמש, שכן היא תאט את השאילתה. אם תמצא ביטוי פנימי המשתמש רק בפונקציות מובנות, תקבל פתרון מהיר ביותר. אם ננסח את הבעיה בצורה טכנית יותר, עליך לקחת את n התווים הראשונים (LEFT(array, n)) ולספור את מספר הפסיקים בתת–המחרוזת. הבעיה היא שרוב פונקציות המחרוזת אינן מטפלות בחזרות או במופעים חוזרים של תת–מחרוזת בתוך מחרוזת. עם זאת, קיימת פונקציה מובנית אחת שכן מטפלת בחזרות – REPLACE. פונקציה זו מחליפה כל מופע של תת–מחרוזת כלשהי (קרא לה oldsubstr) בתוך מחרוזת (קרא לה str) בתת–מחרוזת אחרת (קרא לה newsubstr). אתה קורא לפונקציה עם ארגומנטים אלו בסדר הבא: REPLACE(LEFT(array, n), ',',''). כאן str הוא n התווים הראשונים במערך (LEFT(array, n)), oldsubstr היא פסיק, ו–newsubstr היא מחרוזת ריקה. אנו מחליפים כל

מופע של פסיק בתוך תת–המחרוזת במחרוזת ריקה. כעת, מה ניתן לומר בנוגע להפרש האורך בין תת–המחרוזת המקורית (n) לחדשה? החדשה תהיה כמובן n – num_commas כאשר num_commas הוא מספר הפסיקים ב–str. במילים אחרות n – (n – num_commas) ייתן לך את מספר הפסיקים. הוסף אחד, ותקבל את מיקום האלמנט בתוך המערך. השתמש בפונקציה LEN כדי להחזיר את מספר התווים ב–str לאחר הסרת הפסיקים. להלן הביטוי השלם המחשב את pos:

```
n - LEN(REPLACE(LEFT(array, n), ',', '')) + 1 AS pos
```

שימוש בפונקציה REPLACE לספירת מופעים של מחרוזת בתוך מחרוזת היא טכניקה שימושית.

טיפ: ב– SQL Server 2005, ניתן להשתמש בפונקציה ROW_NUMBER() לחישוב pos:

```
ROW_NUMBER() OVER(PARTITION BY arrid ORDER BY n) AS pos
```

השאילתה הבאה מציגה את הפתרון המלא לבעיה, כולל חישוב המיקום:

```
SELECT arrid,
  n - LEN(REPLACE(LEFT(array, n), ',', '')) + 1 AS pos,
  CAST(SUBSTRING(array, n, CHARINDEX(',', array + ',', n) - n)
       AS INT) AS element
FROM dbo.Arrays
  JOIN dbo.Nums
    ON n <= LEN(array)
    AND SUBSTRING(',' + array, n, 1) = ',';
```

ב– SQL Server 2005, באפשרותך להשתמש ב–CTE רקורסיבי להפרדת אלמנטים ללא הצורך להשתמש בטבלת עזר של מספרים.

```
WITH SplitCTE AS
(
  SELECT arrid, 1 AS pos, 1 AS startpos,
    CHARINDEX(',', array + ',') - 1 AS endpos
  FROM dbo.Arrays
  WHERE LEN(array) > 0

  UNION ALL

  SELECT Prv.arrid, Prv.pos + 1, Prv.endpos + 2,
    CHARINDEX(',', Cur.array + ',', Prv.endpos + 2) - 1
  FROM SplitCTE AS Prv
    JOIN dbo.Arrays AS Cur
      ON Cur.arrid = Prv.arrid
      AND CHARINDEX(',', Cur.array + ',', Prv.endpos + 2) > 0
)
SELECT A.arrid, pos,
  CAST(SUBSTRING(array, startpos, endpos-startpos+1) AS INT) AS element
FROM dbo.Arrays AS A
  JOIN SplitCTE AS S
    ON S.arrid = A.arrid
ORDER BY arrid, pos;
```

ה–CTE מחשב את מיקום ההתחלה והסוף של כל אלמנט. איבר העוגן מחשב את הערכים של האלמנט הראשון בכל מערך. האיבר הרקורסיבי מחשב את הערכים של האלמנטים "הבאים", ומסתיים כאשר לא נמצאים יותר אלמנטים "באים". הטור pos מאותחל עם הקבוע 1, וגדל ב–1 בכל חזרה. השאילתה החיצונית מבצעת join בין טבלת Array ל–CTE, ומחלצת את האלמנטים האינדיבידואליים במערך בהתבסס על מיקומי ההתחלה והסוף המחושבים על ידי ה–CTE. פתרון זה מעט איטי יותר מהקודם, אך יש לו את היתרון שהוא אינו דורש טבלת עזר של מספרים.

פעם שלחתי את החידה הזו לפורום פרטי של מדריכי SQL. אחד המדריכים שלח פתרון מאוד מחוכם שאחד מהקולגות שלו העלה. זה הפתרון:

```
SELECT CAST(arrid AS VARCHAR(10)) AS arrid,
  REPLACE(array, ',', CHAR(10)+CHAR(13)
    + CAST(arrid AS VARCHAR(10)) + SPACE(10)) AS value
FROM dbo.Arrays;
```

ראשית בחן את הפתרון לראות אם הוא מובן לך, ואז הרץ אותו במצב פלט Results To Text. תקבל את הפלט המוצג בטבלה 12–5, אשר "נראה" נכון.

טבלה 12–5: פלט של פתרון לבעיית הפרדת אלמנטים אשר "נראה" נכון

arrid	*value*
A	20
A	22
A	25
A	25
A	14
B	30
B	33
B	28
C	12
C	10
C	8
C	12
C	12
C	13
C	12
C	14
C	10
C	9
D	-4
D	-6
D	-4
D	-2

פתרון זה מחליף כל פסיק בתו newline, (CHAR(10) + CHAR(13)) + array id + 10 spaces. זה נראה נכון כאשר אתה מריץ זאת במצב טקסט, אך זה לא נכון. אם אתה מריץ זאת במצב פלט grid, תראה שהפלט אכן מכיל שורה אחת בלבד לכל מערך.

פעולות סטים

ניתן לחשוב על joins כפעולות **אופקיות** בין טבלאות, המייצרות טבלה וירטואלית המכילה טורים משתי הטבלאות. סעיף זה מכסה פעולות **אנכיות** בין טבלאות, כולל UNION, EXCEPT ו–INTERSECT. כל אזכור של **פעולות סטים** בסעיף זה מתייחס לפעולות אנכיות אלה.

פעולת סט מקבלת שתי טבלאות כקלט, כל אחת תוצאה של הגדרת שאילתה אחרת. לשם הפשטות, אשתמש רק במונח **קלטים (inputs)** בסעיף זה כדי לתאר את טבלאות הקלט של פעולות הסטים.

UNION מחזיר את הסט המאוחד של השורות משני הקלטים, EXCEPT מחזיר את השורות המופיעות בקלט הראשון אך לא בשני, ו–INTERSECT מחזיר שורות המשותפות לשני הקלטים.

ANSI SQL:1999 מגדיר אופרטורים מובנים לכל שלוש פעולות הסט, כל אחד עם שני ניואנסים: אחד יכול אופציונלית להיות מלווה ב–DISTINCT (ברירת המחדל) ואחד מלווה ב–ALL. SQL Server 2000 תמך רק בשתיים מפעולות סטים אלה, UNION ו– UNION ALL. SQL Server 2005 הוסיף תמיכה מובנית לאופרטורי הסט EXCEPT ו–INTERSECT. נכון לעכשיו, SQL Server לא תומך בשימוש האופציונלי של DISTINCT עבור פעולות סטים. זו אינה מגבלה פונקציונלית מכיוון ש–DISTINCT נוסף בעקיפין כאשר אינך מציין ALL. אדון בפתרונות לכל פעולות הסט, עם שני הניואנסים, בשתי הגרסאות.

כמו joins, פעולות סטים אלה תמיד פועלות רק על שני קלטים, ומייצרות טבלה וירטואלית כתוצאה. ייתכן שיקל עליך לקרוא לטבלאות הקלט **שמאלית** ו**ימנית** כמו עם joins, או שאולי יקל עליך יותר להתייחס אליהם כטבלאות קלט **ראשונה** ו**שנייה**.

בטרם אתאר כל פעולת סט בפירוט, הבה נסיר מהדרך מספר סוגיות טכניות לגבי הדרך בה פעולות סטים עובדות.

לשני הקלטים חייב להיות אותו מספר של טורים, וטורים תואמים חייבים להיות מאותו טיפוס נתונים, או לפחות להיות ניתנים להמרה בעקיפין. שמות הטורים של התוצאה נקבעים לפי הקלט הראשון.

פסוקית ORDER BY אינה מותרת בביטויי הטבלה האינדיבידואליים. כל שלבי העיבוד הלוגי האחרים (joins, סינון, קיבוץ וכך הלאה) נתמכים בשאילתות האינדיבידואליות מלבד האפשרות TOP.

בצורה הפוכה, ORDER BY הוא שלב העיבוד הלוגי היחיד הנתמך ישירות בתוצאה הסופית של פעולת סט. אם אתה מציין פסוקית ORDER BY בסוף השאילתה, היא תופעל על סט התוצאה הסופי. אף אחד משלבי העיבוד הלוגי האחרים אינו מותר ישירות על התוצאה של פעולת סט. אספק לכך אלטרנטיבות אחרות בהמשך הפרק.

פעולות סטים עובדות על שורות שלמות משתי טבלאות הקלט. שים לב שכאשר פעולות סטים משוות שורות בין הקלטים, הן מתייחסות ל–NULLs כשווים, ממש כמו ערכים ידועים זהים. בהקשר זה, פעולות סטים אינן דומות למסנני שאילתה (ON, WHERE, HAVING), אשר כפי שתזכור, אינם מתייחסים ל–NULLs כשווים.

UNION

UNION מייצר סט תוצאה המאחד את השורות משני הקלטים. הקטעים הבאים מתארים את ההבדלים בין UNION (DISTINCT מרומז) לבין UNION ALL.

UNION DISTINCT

ציון UNION ללא האפשרות ALL מאחד את השורות משני הקלטים ומפעיל עליהם DISTINCT (במילים אחרות, מסיר שורות כפולות).

לדוגמה, השאילתה הבאה מחזירה את כל המופעים של Country, Region, City המופיעים בטבלת Employees או בטבלת Customers, כשהשורות הכפולות שלהן מוסרות:

```
USE Northwind;

SELECT Country, Region, City FROM dbo.Employees
UNION
SELECT Country, Region, City FROM dbo.Customers;
```

השאילתה מחזירה 71 שורות ייחודיות.

UNION ALL

תוכל לחשוב על UNION ALL כעל UNION ללא הסרת כפילויות. כלומר, תקבל סט תוצאה אחד המכיל את כל השורות משני הקלטים, כולל כפילויות. לדוגמה, השאילתה הבאה מחזירה את כל המופעים של Customer, Region, City משתי הטבלאות:

```
SELECT Country, Region, City FROM dbo.Employees
UNION ALL
SELECT Country, Region, City FROM dbo.Customers;
```

מכיוון שבטבלת Employees 9 שורות ובטבלת Customers 91 שורות, תקבל סט תוצאה עם 100 שורות.

EXCEPT

EXCEPT מאפשר לך לזהות שורות המופיעות בקלט הראשון אך לא בשני.

EXCEPT DISTINCT

EXCEPT DISTINCT מחזיר שורות ייחודיות המופיעות בקלט הראשון אך לא בקלט השני. בשביל להשיג EXCEPT, תוכניתנים משתמשים בדרך כלל בביטוי NOT EXIST, או ב– outer join המסנן שורות חיצוניות בלבד, כפי שהדגמתי קודם בסעיף "Semi Joins". אלא שפתרונות אלו מתייחסים לשני NULLs כשונים זה מזה. למשל, (UK, NULL, London) לא ייחשב כשווה ל– (UK, NULL, London). אם שתי הטבלאות מכילות שורה כזו, **input1 EXCEPT input2** אינה אמורה להחזיר אותה, ואילו פתרונות NOT EXIST ו– outer join כן יחזירו אותה.

בגרסאות SQL Server קודמות ל–2005 לא הייתה תמיכה לאופרטור EXCEPT. לקוד הבא יש פתרון התואם לגרסאות SQL Server קודמות ל– SQL Server 2005 ואשר מקביל לוגית ל– EXCEPT DISTINCT:

```
SELECT Country, Region, City
FROM (SELECT DISTINCT 'E' AS Source, Country, Region, City
      FROM dbo.Employees
      UNION ALL
      SELECT DISTINCT 'C', Country, Region, City
      FROM dbo.Customers) AS UA
GROUP BY Country, Region, City
HAVING COUNT(*) = 1 AND MAX(Source) = 'E';
```

שאילתת הטבלה הנגזרת מאחדת את השורות הייחודיות משני הקלטים, תוך שהיא מקצה זיהוי של קלט המקור לכל שורה (טור תוצאה הנקרא source). שורות מטבלת Employees מקבלות את הזיהוי E, ושורות מטבלת Customers מקבלות את הזיהוי C. השאילתה החיצונית מקבצת את השורות לפי Country, Region, City. הפסוקית HAVING שומרת רק קבוצות בעלות שורה אחת (כלומר שהשורה הופיעה רק באחד מהקלטים) וזיהוי מקור מקסימלי E (כלומר שהשורה הייתה מטבלת Employees). לוגית, זהו EXCEPT DISTINCT.

ב– SQL Server 2005, הדברים מעט יותר פשוטים:

```
SELECT Country, Region, City FROM dbo.Employees
EXCEPT
SELECT Country, Region, City FROM dbo.Customers;
```

שים לב שמתוך שלוש פעולות הסטים, רק EXCEPT אינו סימטרי. כלומר, **input1 EXCEPT input2** אינו אותו דבר כמו **input2 EXCEPT input1**.

למשל, השאילתה שזה עתה הוצגה מחזירה את שתי הערים המופיעות בטבלת Employees אך לא בטבלת Customers. השאילתה הבאה מחזירה 66 ערים המופיעות בטבלת Customers אך לא בטבלת Employees:

```
SELECT Country, Region, City FROM dbo.Customers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees;
```

EXCEPT ALL

EXCEPT ALL מעט יותר מורכב מאשר EXCEPT DISTINCT והוא עדיין לא יושם ב– SQL Server. מלבד זה שהוא מתעניין בקיומה של שורה, הוא מתעניין גם במספר המופעים של כל שורה. נאמר שאתה מבקש את התוצאה של **input1 EXCEPT ALL input2**. אם שורה מופיעה n פעמים ב–input1 ו–m פעמים ב–input2 (הן n והן m יכולים להיות גדולים או שווים לאפס), היא תופיע MAX(0, n-m) פעמים בפלט. כלומר, אם n גדול מ–m, השורה תופיע n-m פעמים בתוצאה; אחרת, היא לא תופיע בתוצאה כלל.

השאילתה הבאה מדגימה כיצד ניתן להשיג EXCEPT ALL על ידי שימוש בפתרון התואם לגרסאות קודמות ל– SQL Server 2005:

```
SELECT Country, Region, City
FROM (SELECT Country, Region, City,
        MAX(CASE WHEN Source = 'E' THEN Cnt ELSE 0 END) ECnt,
        MAX(CASE WHEN Source = 'C' THEN Cnt ELSE 0 END) CCnt
      FROM (SELECT 'E' AS Source,
              Country, Region, City, COUNT(*) AS Cnt
            FROM dbo.Employees
            GROUP BY Country, Region, City

            UNION ALL

            SELECT 'C', Country, Region, City, COUNT(*)
            FROM dbo.Customers
            GROUP BY Country, Region, City) AS UA
      GROUP BY Country, Region, City) AS P
JOIN dbo.Nums
  ON n <= ECnt - CCnt;
```

לטבלה הנגזרת UA יש שורה לכל שורת מקור ייחודית מכל קלט, בצירוף זיהוי המקור (E עבור Employees, C עבור Customers) ומספר הפעמים (Cnt) שהיא מופיעה במקור.

השאילתה המייצרת את הטבלה הנגזרת P מקבצת את השורות מטבלת UA לפי Country, Region ו–City. היא משתמשת בשני ביטויי MAX(CASE...) כדי להחזיר את ספירת הכפילויות משני הקלטים באותה שורת תוצאה, וקוראת להן ECnt ו–CCnt. זוהי שיטה לסיבוב נתונים על ציר, ואדון בה בפירוט בפרק 6. בנקודה זו, לכל מופע ייחודי של Country, Region, City יש שורה יחידה ב–P, בצירוף מספר הכפילויות שהיו לה בכל קלט. לבסוף, השאילתה החיצונית מאחדת את O עם Nums כדי לייצר כפילויות. תנאי ה–join הוא n <= ECnt - CCnt. אם תחשוב על כך, תקבל בדיוק את אותו מספר כפילויות המוכתב על ידי EXCEPT ALL. כלומר, אם ECnt – CCnt גדול מ–0, תקבל מספר זה של כפילויות; אחרת, לא תקבל כפילויות כלל.

על אף שאין לך אופרטור מובנה עבור EXCEPT ALL ב– SQL Server 2005, תוכל בקלות לייצר מקבילה לוגית על ידי שימוש ב–EXCEPT ובפונקציית ROW_NUMBER. להלן הפתרון:

```
WITH EXCEPT_ALL
AS
(
  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER     BY Country, Region, City) AS rn,
    Country, Region, City
    FROM dbo.Employees

  EXCEPT

  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER     BY Country, Region, City) AS rn,
      Country, Region, City
  FROM dbo.Customers
)
SELECT Country, Region, City
FROM EXCEPT_ALL;
```

כדי להבין פתרון זה, אני מציע שראשית תדגיש את השאילתות (קטעים) שבתוכו ותריץ אותן בנפרד. דבר זה יאפשר לך לבחון את תוצאות הביניים ולקבל מושג טוב יותר על מה שהפסקה הבאה מנסה להסביר.

הקוד ראשית מקצה מספרי שורה לשורות של כל קלט, וחוצץ אותן לפי כל רשימת הטורים. מספרי השורה ימספרו את השורות הכפולות בתוך הקלט. למשל, שורה המופיעה חמש פעמים

בטבלת Employees ושלוש פעמים בטבלת Customers, תקבל מספרי שורה 1 עד 5 בקלט הראשון, ומספרי שורה 1 עד 3 בקלט השני. אז אתה מפעיל **input1 EXCEPT input2** ומקבל שורות (כולל הטור rn) המופיעות ב–input1 אך לא ב–input2. אם שורה R מופיעה 5 פעמים ב–input1 ו–3 פעמים ב–input2, תקבל את התוצאה הבאה:

```
{(R, 1), (R, 2), (R, 3), (R, 4), (R, 5)}
EXCEPT
{(R, 1), (R, 2), (R, 3)}
```

וזו מפיקה את התוצאה הבאה:

```
{(R, 4), (R, 5)}
```

במילים אחרות, R תופיע בתוצאה בדיוק כמספר הפעמים שהיה מתקבל על ידי EXCEPT ALL. עטפתי את הלוגיקה הזו ב–CTE כדי להחזיר רק את רשימת הטורים המקורית ללא מספר השורה, שזה מה שהיה עושה EXCEPT ALL.

INTERSECT

INTERSECT מחזיר שורות המופיעות בשני הקלטים.

בשביל להשיג INTERSECT, תוכניתנים לרוב משתמשים בביטוי EXIST או ב– inner join, כפי שהדגמתי קודם בסעיף "Semi Join". אלא, שכפי שהסברתי קודם, פתרונות אלו מתייחסים לשני NULLs כשונים זה מזה, ואילו פעולות סטים אמורות להתייחס אליהם כשווים.

תמיכה באופרטור INTERSECT הוצגה ב– SQL Server 2005, אך רק בווריאציה עם DISTINCT מרומז.

INTERSECT DISTICNT

כדי להשיג את המקבילה הלוגית של INTERSECT DISTINCT טרום SQL Server 2005, תוכל להשתמש בפתרון הבא:

```
SELECT Country, Region, City
FROM (SELECT DISTINCT Country, Region, City FROM dbo.Employees
      UNION ALL
      SELECT DISTINCT Country, Region, City FROM dbo.Customers) AS UA
GROUP BY Country, Region, City
HAVING COUNT(*) = 2;
```

הטבלה הנגזרת מבצעת UNION ALL בין שורות ייחודיות לשני הקלטים. השאילתה החיצונית מקבצת את התוצאה המאוחדת לפי Country, Region ו–City, ומחזירה רק קבוצות להן שני מופעים. במילים אחרות, השאילתה מחזירה רק שורות ייחודיות המופיעות בשני הקלטים, שזו ההגדרה של INTERSECT DISTINCT.

ב– SQL Server 2005, אתה פשוט משתמש באופרטור INTERSECT כלהלן:

```
SELECT Country, Region, City FROM dbo.Employees
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

INTERSECT ALL

כמו EXCEPT ALL, גם INTERSECT ALL בוחן מופעים מרובים של שורות. אם שורה R מופיעה n פעמים בטבלת קלט אחת ו–m פעמים בשנייה, היא צריכה להופיע MIN(n,m) פעמים בתוצאה.

השיטות להשגת INTERSECT ALL דומות מהרבה בחינות לשיטות המשמשות להשגת EXCEPT ALL. למשל, להלן פתרון טרום SQL Server 2005 להשגת INTERSECT ALL:

```
SELECT Country, Region, City
FROM (SELECT Country, Region, City, MIN(Cnt) AS MinCnt
      FROM (SELECT Country, Region, City, COUNT(*) AS Cnt
            FROM dbo.Employees
            GROUP BY Country, Region, City

            UNION ALL

            SELECT Country, Region, City, COUNT(*)
            FROM dbo.Customers
            GROUP BY Country, Region, City) AS UA
      GROUP BY Country, Region, City
      HAVING COUNT(*) > 1) AS D
  JOIN dbo.Nums
    ON n <= MinCnt;
```

ל–UA יש את ה– UNION ALL של שורות ייחודיות מכל קלט בצירוף מספר המופעים שהיה להן במקור. השאילתה מול UA המייצרת את הטבלה הנגזרת D מקבצת את השורות לפי Country, Region ו–City. הפסוקית HAVING מסננת רק שורות המופיעות בשני הקלטים (COUNT(*) > 1), ומחזירה את הספירה המינימלית שלהם (MinCnt). זהו מספר

הפעמים שהשורה צריכה להופיע בפלט. כדי לייצר מספר כזה של העתקים, השאילתה החיצונית מצליבה את D עם Nums בהתבסס על n <= MinCnt.

הפתרון ל– INTERSECT ALL ב– SQL Server 2005 זהה לזה עבור EXCEPT ALL מלבד הבדל ברור אחד – השימוש באופרטור INTERSECT במקום EXCEPT:

```
WITH INTERSECT_ALL
AS
(
  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER     BY Country, Region, City) AS rn,
    Country, Region, City
  FROM dbo.Employees

  INTERSECT

  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER     BY Country, Region, City) AS rn,
    Country, Region, City
    FROM dbo.Customers
)
SELECT Country, Region, City
FROM INTERSECT_ALL;
```

קדימות של פעולות סטים

לפעולת הסטים INTERSECT יש קדימות גבוהה יותר מאשר לאחרות. בשאילתה המשלבת מספר פעולות סטים, INTERSECT מוערכת ראשונה. מעבר לכך, פעולות סטים מוערכות משמאל לימין. היוצא מן הכלל הוא שסוגריים תמיד ראשונות בקדימות, כך שעל ידי שימוש בסוגריים יש לך שליטה מלאה על הסדר הלוגי של הערכת פעולות סטים.

למשל, בשאילתה הבאה, INTERSECT מוערכת ראשונה למרות שהיא מופיעה שנייה:

```
SELECT Country, Region, City FROM dbo.Suppliers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

המשמעות של השאילתה היא: החזר ערים של ספקים שאינן מופיעות בהצלבה של ערי עובדים וערי לקוחות.

אך אם אתה משתמש בסוגריים, באפשרותך לשנות את סדר ההערכה:

```
(SELECT Country, Region, City FROM dbo. Suppliers
 EXCEPT
 SELECT Country, Region, City FROM dbo.Employees)
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

המשמעות של שאילתה זו היא: החזר ערים של ספקים שאינן ערי עובדים וכמו כן הינן ערי לקוחות.

שימוש ב–INTO עם פעולות סטים

אם ברצונך לכתוב משפט SELECT INTO כאשר אתה משתמש בפעולות סטים, הגדר את הפסוקית INTO מייד לפני הפסוקית FROM בקלט הראשון. למשל, תוכל לראות כיצד אתה ממלא טבלה זמנית #T בנתוני תוצאת אחת מהשאילתות הקודמות:

```
SELECT Country, Region, City INTO #T FROM dbo.Suppliers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

עקיפת שלבים לוגיים לא נתמכים

כפי שהזכרתי קודם, שלבי עיבוד לוגיים אחרים מאשר מיון (joins, סינון, קיבוץ, TOP וכך הלאה) אינם מותרים ישירות על התוצאה של פעולת סט. מגבלה זו ניתנת לעקיפה בקלות על ידי שימוש בטבלה נגזרת או ב–CTE כלהלן:

```
SELECT DISTINCT TOP ...
FROM (<set operation query>) AS D
JOIN | PIVOT | APPLY ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
```

למשל, השאילתה הבאה (המפיקה את הפלט המוצג בטבלה 13–5) אומרת לך את מספר הערים בכל ארץ, שיש בהן לקוחות או עובדים:

```
SELECT Country, COUNT(*) AS NumCities
FROM (SELECT Country, Region, City FROM dbo.Employees
      UNION
      SELECT Country, Region, City FROM dbo.Customers) AS U
GROUP BY Country;
```

טבלה 13–5: מספר הערים בכל ארץ שיש בהן לקוחות או עובדים

Country	*NumCities*
Argentina	1
Austria	2
Belgium	2
Brazil	4
Canada	3
Denmark	2
Finland	2
France	9
Germany	11
Ireland	1
Italy	3
Mexico	1
Norway	1
Poland	1
Portugal	1
Spain	3
Sweden	2
Switzerland	2
UK	2
USA	14
Venezuela	4

באופן דומה, תוכל לעקוף את המגבלות על השאילתות האינדיבידואליות המשמשות כקלטים לפעולת הסט. כל קלט יכול להיכתב כשאילתת SELECT פשוטה מטבלה נגזרת או CTE, כאשר אתה משתמש באלמנטים ה"אסורים" בטבלה הנגזרת או בביטוי ה–CTE.

לדוגמה, השאילתה הבאה מחזירה את שתי ההזמנות האחרונות עבור עובדים 3 ו–5, ומפיקה את הפלט המוצג בטבלה 14–5:

```
SELECT EmployeeID, OrderID, OrderDate
FROM (SELECT TOP 2 EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE EmployeeID = 3
      ORDER BY OrderDate DESC, OrderID DESC) AS D1

UNION ALL

SELECT EmployeeID, OrderID, OrderDate
FROM (SELECT TOP 2 EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE EmployeeID = 5
      ORDER BY OrderDate DESC, OrderID DESC) AS D2;
```

טבלה 14–5: שתי ההזמנות האחרונות של עובדים 3 ו–5

EmployeeID	*OrderID*	*OrderDate*
3	11063	1998-04-30 00:00:00.000
3	11057	1998-04-29 00:00:00.000
5	11043	1998-04-22 00:00:00.000
5	10954	1998-03-17 00:00:00.000

באשר למגבלה על מיון הקלטים האינדיבידואליים, נניח שעליך למיין כל קלט בנפרד. למשל, ברצונך להחזיר הזמנות שבוצעו על ידי הלקוח ALFKI וכן הזמנות המטופלות על ידי עובד 3. באשר למיון השורות בפלט, ברצונך שההזמנות של הלקוח ALFKI יופיעו ראשונות, ממוינות בסדר OrderDate יורד, ואז הזמנות שטופלו על ידי עובד 3, ממוינות בסדר OrderID עולה. כדי להשיג זאת, אתה יוצר טור (SortCol) עם הקבוע 1 עבור הקלט הראשון (לקוח ALFKI), ו–2 עבור השני (עובד 3). אז אתה יוצר טבלה נגזרת (קרא לה U) מתוך ה– UNION ALL בין השניים. בשאילתה החיצונית, ראשית מיין לפי SortCol, ואז לפי ביטוי CASE לכל סט. ביטוי ה–CASE יחזיר את הערך הרלוונטי בהתבסס על סט המקור; אחרת, הוא מחזיר NULL, שלא ישפיע על המיון. להלן שאילתת הפתרון המפיקה את הפלט (בצורה מקוצרת) המופיעה בטבלה 15–5:

```
SELECT EmployeeID, CustomerID, OrderID, OrderDate
FROM (SELECT 1 AS SortCol, CustomerID, EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE CustomerID = N'ALFKI'

      UNION ALL

      SELECT 2 AS SortCol, CustomerID, EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE EmployeeID = 3) AS U
ORDER BY SortCol,
  CASE WHEN SortCol = 1 THEN OrderID END,
  CASE WHEN SortCol = 2 THEN OrderDate END DESC;
```

טבלה 15–5: מיון כל קלט בנפרד (בצורה מקוצרת)

EmployeeID	*CustomerID*	*OrderID*	*OrderDate*
6	ALFKI	10643	1997-08-25 00:00:00.000
4	ALFKI	10692	1997-10-03 00:00:00.000
4	ALFKI	10702	1997-10-13 00:00:00.000
1	ALFKI	10835	1998-01-15 00:00:00.000
1	ALFKI	10952	1998-03-16 00:00:00.000
3	ALFKI	11011	1998-04-09 00:00:00.000
3	HUNGO	11063	1998-04-30 00:00:00.000
3	NORTS	11057	1998-04-29 00:00:00.000
3	HANAR	11052	1998-04-27 00:00:00.000
3	GOURL	11049	1998-04-24 00:00:00.000
3	CHOPS	11041	1998-04-22 00:00:00.000
3	QUICK	11021	1998-04-14 00:00:00.000
…			

סיכום

כיסיתי היבטים רבים של joins ופעולות סטים והדגמתי שיטות חדשות ושימושיות.

זכור שהתחביר הישן הייחודי ל–T-SQL עבור joins אינו נתמך יותר ויעבוד רק במצב של תאימות אחורה. בו בזמן, סוגים אחרים של joins המשתמשים בתחביר ANSI SQL:1989 נתמכים באופן מלא, שכן תחביר זה הוא עדיין חלק מהתקן – למרות שכאשר משתמשים בתחביר הישן עבור inner joins, קיים סיכון של קבלת מכפלה קרטזית כאשר אתה שוכח לציין פסוקית WHERE.

SQL Server 2005 מציג אופרטורים מובנים עבור EXCEPT ו–INTERSECT. הוא מספק גם כלים אחרים המאפשרים פתרונות פשוטים להשגת EXCEPT ALL ו– INTERSECT ALL.

6

פונקציות צבירה של נתונים וסיבוב על ציר

פרק זה עוסק בשיטות שונות של סיכומי–נתונים (data-aggregation), כולל הפסוקית החדשה OVER, שוברי–שוויון, פונקציית צבירה (אגרגציה) רצה, סיבוב על ציר, פונקציות צבירה מוגדרות–משתמש, היסטוגרמות, גורמי הקבצה (grouping factors), והאופציות CUBE ו–ROLLUP.

בפרק זה אשתמש שוב בשיטות שהצגתי קודם לכן וגם אציג שיטות חדשות לפתרון הבעיות.

לוגיקה תהווה כמובן חלק בלתי נפרד מהפתרונות. זכור שבלב כל בעיית שאילתות מונחת חידת היגיון.

הפסוקית OVER

הפסוקית OVER מאפשרת לבקש חישובים מבוססי–חלון – כלומר, החישוב מבוצע על חלון שלם של ערכים. בפרק 4, תיארתי בפירוט כיצד להשתמש בפסוקית OVER עם פונקציות ניתוחי הדירוג החדשות. Microsoft SQL Server 2005 מציג גם תמיכה בפסוקית OVER עם פונקציות צבירה (אגרגציות) סקלאריות; אך נכון לעכשיו ניתן להשתמש בהן רק עם הפסוקית PARTITION BY. בשאיפה, גרסאות עתידיות של SQL Server יתמכו גם באלמנטים האחרים של פונקציות צבירה חלוניות לפי תקן ANSI, כולל הפסוקיות ORDER BY ו–ROWS.

המטרה של שימוש בפסוקית OVER עם פונקציות צבירה סקלאריות היא לחשב, עבור כל שורה, צבירה המתבססת על חלון של ערכים מעבר לטווח השורה – ולעשות זאת מבלי להשתמש בפסוקית GROUP BY בשאילתה. במילים אחרות, הפסוקית OVER מאפשרת להוסיף חישובי צבירה לתוצאות של שאילתות לא מקובצות. יכולת זו מספקת חלופה לבקשת נתוני צבירה עם תת–שאילתות, במקרה בו אתה נדרש לכלול בתוצאות שלך הן מאפיינים של שורות בסיס והן נתוני צבירה.

כתזכורת, בפרק 5 הצגתי בעיה בה נדרשת לחשב שני נתוני צבירה (אגרגציה) לכל שורת מכירה: האחוז שהשורה תרמה לסך כמות המכירה וההפרש בין כמות המכירה של השורה והכמות הממוצעת של כל המכירות. הצגתי את השאילתה המשופרת שלהלן בה השתמשתי ב– cross join בין טבלת הבסיס לטבלה הנגזרת של נתוני הצבירה, במקום להשתמש במספר תת–שאילתות:

```
SET NOCOUNT ON;
USE pubs;

SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / sumqty * 100 AS DECIMAL(5, 2)) AS per,
  CAST(qty - avgqty AS DECIMAL(9, 2)) as diff
FROM dbo.sales,
  (SELECT SUM(qty) AS sumqty, AVG(1.*qty) AS avgqty
   FROM dbo.sales) AS AGG;
```

שאילתה זו יצרה את הפלט המוצג בטבלה 1–6.

טבלה 1–6: אחוז מסך המכירות והפרש מממוצע

stor_id	*ord_num*	*title_id*	*ord_date*	*qty*	*per*	*diff*
6380	6871	BU1032	1994-09-14	5	1.01	-18.48
6380	722a	PS2091	1994-09-13	3	0.61	-20.48
7066	A2976	PC8888	1993-05-24	50	10.14	26.52
7066	QA7442.3	PS2091	1994-09-13	75	15.21	51.52
7067	D4482	PS2091	1994-09-14	10	2.03	-13.48
7067	P2121	TC3218	1992-06-15	40	8.11	16.52
7067	P2121	TC4203	1992-06-15	20	4.06	-3.48
7067	P2121	TC7777	1992-06-15	20	4.06	-3.48
7131	N914008	PS2091	1994-09-14	20	4.06	-3.48
7131	N914014	MC3021	1994-09-14	25	5.07	1.52
7131	P3087a	PS1372	1993-05-29	20	4.06	-3.48
7131	P3087a	PS2106	1993-05-29	25	5.07	1.52
7131	P3087a	PS3333	1993-05-29	15	3.04	-8.48
7131	P3087a	PS7777	1993-05-29	25	5.07	1.52
7896	QQ2299	BU7832	1993-10-28	15	3.04	-8.48

stor_id	ord_num	title_id	ord_date	qty	per	diff
7896	TQ456	MC2222	1993-12-12	10	2.03	-13.48
7896	X999	BU2075	1993-02-21	35	7.10	11.52
8042	423LL922	MC3021	1994-09-14	15	3.04	-8.48
8042	423LL930	BU1032	1994-09-14	10	2.03	-13.48
8042	P723	BU1111	1993-03-11	25	5.07	1.52
8042	QA879.1	PC1035	1993-05-22	30	6.09	6.52

הרצון לחישוב שני נתוני צבירה בטבלה נגזרת אחת במקום בשתי תת–שאילתות נפרדות נבע מהעובדה שכל תת–שאילתה ניגשה לטבלה/אינדקס, בעוד שהטבלה הנגזרת חישבה את נתוני הצבירה בעזרת שימוש בסריקה יחידה של הנתונים.

באופן דומה, ניתן לחשב מספר נתוני צבירה בעזרת שימוש באותה פסוקית OVER, ו– SQL Server יסרוק את נתוני המקור הנדרשים פעם אחת בלבד בשביל כולם. להלן הדרך בה תשתמש בפסוקית OVER כדי לענות על אותה דרישה:

```
SELECT stor_id, ord_num, title_id,
   CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
   CAST(1.*qty / SUM(qty)   OVER() * 100 AS DECIMAL(5, 2)) AS per,
   CAST(qty - AVG(1.*qty)   OVER() AS DECIMAL(9, 2)) AS diff
FROM dbo.sales;
```

שים לב: בפרק 4, תיארתי את הפסוקית PARTITION BY, שיש בה שימוש בפונקציות חלוניות, ובהן גם פונקציות צבירה חלוניות. פסוקית זו היא אופציונלית. כאשר אינה מוגדרת, הצבירה כולה מתבססת על כל הקלט ולא מחושבת בנפרד בכל מחיצה.

כאן, מכיוון שלא הגדרתי פסוקית PARTITION BY, נתוני הצבירה היו מחושבים בהתבסס על הקלט כולו. לוגית, SUM(qty) OVER() מקביל כאן לתת–השאילתה (SELECT SUM(qty) FROM dbo.sales). פיסית, הסיפור שונה לגמרי. כתרגיל, תוכל להשוות את תוכנית העבודה של שתי השאילתות הבאות, כל אחת מבקשת מספר שונה של נתוני צבירה על ידי שימוש באותה פסוקית OVER:

```
SELECT stor_id, ord_num, title_id,
  SUM(qty) OVER() AS sumqty
FROM dbo.sales;

SELECT stor_id, ord_num, title_id,
  SUM(qty)    OVER() AS sumqty,
```

```
  COUNT(qty)  OVER() AS cntqty,
  AVG(qty)    OVER() AS avgqty,
  MIN(qty)    OVER() AS minqty,
  MAX(qty)    OVER() AS maxqty
FROM dbo.sales;
```

תמצא ששתי התוכניות זהות כמעט לחלוטין, כשההבדל היחיד הוא שהאופרטור היחיד Stream Aggregate מחשב מספר שונה של נתוני צבירה לכל אחת. עלות השאילתות זהה. מצד שני, השווה את תוכניות העבודה של שתי השאילתות הבאות, כל אחת מבקשת מספר שונה של נתוני צבירה (אגרגציה) תוך שימוש בתת–שאילתות:

```
SELECT stor_id, ord_num, title_id,
  (SELECT SUM(qty) FROM dbo.sales) AS sumqty
FROM dbo.sales;

SELECT stor_id, ord_num, title_id,
  (SELECT SUM(qty)   FROM dbo.sales) AS sumqty,
  (SELECT COUNT(qty) FROM dbo.sales) AS cntqty,
  (SELECT AVG(qty)   FROM dbo.sales) AS avgqty,
  (SELECT MIN(qty)   FROM dbo.sales) AS minqty,
  (SELECT MAX(qty)   FROM dbo.sales) AS maxqty
FROM dbo.sales;
```

תמצא שיש להן תוכניות שונות, כשהשנייה יקרה יותר, שכן היא סורקת שוב את נתוני המקור לכל נתון צבירה.

יתרון נוסף של הפסוקית OVER הוא, שהיא מאפשרת קוד קצר ופשוט יותר. דבר זה בולט במיוחד כאשר עליך לחשב נתוני צבירה במחיצות. בשימוש ב–OVER, עליך פשוט להגדיר פסוקית PARTITION BY. בשימוש בתת–שאילתות, עליך לקשר בין השאילתה הפנימית לחיצונית, תוך שאתה הופך את השאילתה לארוכה ומורכבת יותר.

כדוגמה לשימוש בפסוקית PARTITION BY, השאילתה הבאה מחשבת את אחוז הכמות מתוך הסך בחנות ואת ההפרש מממוצע החנות, ומפיקה את הפלט המוצג בטבלה 2–6:

```
SELECT stor_id, ord_num, title_id,
  CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
  CAST(1.*qty / SUM(qty) OVER(PARTITION BY stor_id) * 100
    AS DECIMAL(5, 2)) AS per,
  CAST(qty - AVG(1.*qty) OVER(PARTITION BY stor_id)
    AS DECIMAL(9, 2)) AS diff
FROM dbo.sales
ORDER BY stor_id;
```

טבלה 2–6: אחוז מכירות מסך החנות והפרש ממוצע החנות

stor_id	*ord_num*	*title_id*	*ord_date*	*qty*	*per*	*diff*
6380	6871	BU1032	1994-09-14	5	62.50	1.00
6380	722a	PS2091	1994-09-13	3	37.50	-1.00
7066	A2976	PC8888	1993-05-24	50	40.00	-12.50
7066	QA7442.3	PS2091	1994-09-13	75	60.00	12.50
7067	D4482	PS2091	1994-09-14	10	11.11	-12.50
7067	P2121	TC3218	1992-06-15	40	44.44	17.50
7067	P2121	TC4203	1992-06-15	20	22.22	-2.50
7067	P2121	TC7777	1992-06-15	20	22.22	-2.50
7131	N914008	PS2091	1994-09-14	20	15.38	-1.67
7131	N914014	MC3021	1994-09-14	25	19.23	3.33
7131	P3087a	PS1372	1993-05-29	20	15.38	-1.67
7131	P3087a	PS2106	1993-05-29	25	19.23	3.33
7131	P3087a	PS3333	1993-05-29	15	11.54	-6.67
7131	P3087a	PS7777	1993-05-29	25	19.23	3.33
7896	QQ2299	BU7832	1993-10-28	15	25.00	-5.00
7896	TQ456	MC2222	1993-12-12	10	16.67	-10.00
7896	X999	BU2075	1993-02-21	35	58.33	15.00
8042	423LL922	MC3021	1994-09-14	15	18.75	-5.00
8042	423LL930	BU1032	1994-09-14	10	12.50	-10.00
8042	P723	BU1111	1993-03-11	25	31.25	5.00
8042	QA879.1	PC1035	1993-05-22	30	37.50	10.00

בקצרה, הפסוקית OVER מאפשרת שאילתות קצרות ומהירות יותר.

שוברי–שוויון

בסעיף זה אני מעוניין להציג שיטה חדשה המבוססת על נתוני פונקציות צבירה לפתרון בעיות שובר–שוויון, בהן התחלתי לדון בפרק 4. אשתמש בדוגמה זהה לזו בה השתמשתי שם – החזרת ההזמנה החדשה ביותר לכל עובד – תוך שימוש בשילובים שונים של מאפייני שובר–שוויון המזהים ייחודית מיון לכל עובד. זכור שהביצועים של הפתרונות המשתמשים בתת–שאילתות תלויים בצורה רבה מאוד באינדקסים. כלומר, יש צורך באינדקס על

טור המחיצה, טור המיון ומאפייני שובר–השוויון. אך בפועל, לא תמיד קיימת האפשרות להוסיף אינדקסים ככל שתרצה. הפתרונות מבוססי–תת–השאילתות יסבלו מאוד מבחינת ביצועים כתוצאה מהיעדר אינדקסים מתאימים. כאשר תשתמש בשיטות של פונקציות צבירה תמצא שהפתרון הוא בעל ביצועים טובים אפילו כשלא קיים אינדקס אופטימלי – למעשה, אפילו כשלא קיים אינדקס טוב.

הבה נתחיל בשימוש ב–MAX(OrderID) כשובר–השוויון. להזכירך, אתה מעוניין בהזמנה האחרונה של כל עובד, תוך שימוש ב– MAX(OrderID) כשובר השוויון. לכל הזמנה עליך להחזיר EmployeeID, OrderDate, OrderID, CustomerID ו–RequiredDate.

שיטת פונקציית הצבירה לפתרון הבעיה מפעילה את הרעיון הלוגי המופיע להלן בקטע דמוי–קוד:

```
SELECT EmployeeID, MAX(OrderDate, OrderID, CustomerID, RequiredDate)
FROM dbo.Orders
GROUP BY EmployeeID;
```

יכולת כזו לא קיימת ב–T-SQL, כך שאל תנסה להריץ את השאילתה. הרעיון כאן הוא לייצר שורה לכל עובד, עם MAX(OrderDate) ו– MAX(OrderID) – שובר–השוויון – מבין ההזמנות בעלות OrderDate מקסימלי.

מכיוון שהשילוב EmployeeID, OrderDate ו–OrderID הוא כבר ייחודי, כל המאפיינים האחרים (CustomerID, RequiredDate) פשוט מוחזרים מהשורה הנבחרת. מכיוון ש–MAX של יותר ממאפיין אחד לא קיים ב–T-SQL, עליך לחקות זאת בדרך כלשהי, ובאפשרותך לעשות זאת על ידי שרשור כל המאפיינים ליצירת ערך קלט יחידני לפונקציה MAX, ואז בשאילתה חיצונית, חילוץ בחזרה של האלמנטים הבודדים.

השאלה היא זו: באיזו שיטה עליך להשתמש לשרשור המאפיינים? הטריק הוא להשתמש במחרוזת ברוחב קבוע לכל מאפיין ולהמיר את המאפיינים בדרך שלא תשנה את התנהגות המיון. כאשר עוסקים אך ורק במספרים חיוביים, ניתן להשתמש בחישוב אריתמטי למיזוג ערכים. למשל, נניח שיש לך את המספרים m ו–n, לכל אחד תחום תקין של 1 עד 999. כדי למזג את m ו–n, השתמש בנוסחה הבאה: m*1000 + n AS r. כדי לחלץ מאוחר יותר את החלקים הבודדים, השתמש ב–r מחולק ב–1000 כדי לקבל את m, והשתמש ב–r modulo 1000 כדי לקבל את n. עם זאת, במקרים רבים סביר שיהיה עליך לשרשר ערכים לא–מספריים, כך שלא תוכל להשתמש בשרשור אריתמטי. ייתכן שתרצה לשקול המרת כל הערכים למחרוזות תווים בעלות רוחב קבוע (CHAR(n)/NCHAR(n)) או למחרוזות בינאריות בעלות רוחב קבוע (BINARY(n)).

להלן דוגמה להחזרת ההזמנה בעלת MAX(OrderDate) לכל עובד, תוך שימוש ב– MAX(OrderID) כשובר–שוויון, על ידי שימוש בשרשור בינארי:

```
USE Northwind;

SELECT EmployeeID,
  CAST(SUBSTRING(binstr, 1, 8)      AS DATETIME) AS OrderDate,
  CAST(SUBSTRING(binstr, 9, 4)      AS INT)      AS OrderID,
  CAST(SUBSTRING(binstr, 13, 10)    AS NCHAR(5)) AS CustomerID,
  CAST(SUBSTRING(binstr, 23, 8)     AS DATETIME) AS RequiredDate
FROM (SELECT EmployeeID,
        MAX(CAST(OrderDate          AS BINARY(8))
              + CAST(OrderID AS BINARY(4))
              + CAST(CustomerID     AS BINARY(10))
              + CAST(RequiredDate   AS BINARY(8))) AS binstr
      FROM dbo.Orders
      GROUP BY EmployeeID) AS D;
```

הטבלה הנגזרת D מכילה את המחרוזת המשורשרת המקסימלית לכל עובד. שים לב שכל ערך הומר למחרוזת המתאימה בעלת הגודל הקבוע לפני השרשור, על פי טיפוס הנתונים שלו (DATETIME – 8 בתים, INT – 4 בתים, וכך הלאה).

שים לב: כאשר ממירים מספרים למחרוזות בינאריות, רק ערכים לא–שליליים יישמרו על התנהגות המיון המקורית שלהם. באשר למחרוזות תווים, המרתם לערכים בינאריים גורמת להם להשתמש בהתנהגות מיון דומה לסדר מיון בינארי.

היתרון האמיתי בפתרון זה הוא שהוא סורק את הנתונים פעם אחת בלבד, ללא תלות בקיומו של אינדקס טוב. אם יש אינדקס, סביר להניח שתקבל סריקה ממוינת של האינדקס וסיכום מבוסס–מיון. אם לא קיים, כמו במקרה הנוכחי, סביר להניח שתקבל סיכום מבוסס–hash, כפי שניתן לראות בתרשים 1–6.

תרשים 1–6: תוכנית עבודה לשאילתת שובר–שוויון

הדברים הופכים להיות מתוחכמים יותר כאשר למאפייני טורי המיון ושובר–השוויון יש כיווני מיון שונים בתוכם. למשל, נניח ששובר–השוויון היה MIN(OrderID). במקרה זה, תצטרך להפעיל MAX ל–OrderDate, ו–MIN ל–OrderID. ישנו פתרון לוגי כאשר המאפיין עם הכיוון ההפוך הוא מספרי. נניח שעליך לחשב את ערך ה–MIN של טור

המספר השלם הלא–שלילי n, תוך שימוש ב–MAX בלבד. ניתן לבצע זאת על ידי שימוש ב: <maxint> - MAX(<maxint> - n).

השאילתה הבאה משלבת את השיטה הלוגית הזו:

```
SELECT EmployeeID,
  CAST(SUBSTRING(binstr, 1, 8)     AS DATETIME) AS OrderDate,
  2147483647 - CAST(SUBSTRING(binstr, 9, 4) AS INT) AS OrderID,
  CAST(SUBSTRING(binstr, 13, 10)   AS NCHAR(5)) AS CustomerID,
  CAST(SUBSTRING(binstr, 23, 8)    AS DATETIME) AS RequiredDate
FROM (SELECT EmployeeID,
        MAX(CAST(OrderDate         AS BINARY(8))
              + CAST(2147483647 - OrderID AS BINARY(4))
              + CAST(CustomerID    AS BINARY(10))
              + CAST(RequiredDate  AS BINARY(8))) AS binstr
      FROM dbo.Orders
      GROUP BY EmployeeID) AS D;
```

כמובן שבאפשרותך לשחק בשובר–השוויון בו אתה משתמש בכל דרך שתבחר. למשל, להלן השאילתה שתחזיר את ההזמנה האחרונה לכל עובד, תוך שימוש ב– MAX(RequiredDate), MAX(OrderID) כשובר–שוויון:

```
SELECT EmployeeID,
  CAST(SUBSTRING(binstr, 1, 8)   AS DATETIME) AS OrderDate,
  CAST(SUBSTRING(binstr, 9, 8)   AS DATETIME) AS RequiredDate,
  CAST(SUBSTRING(binstr, 17, 4)  AS INT)      AS OrderID,
  CAST(SUBSTRING(binstr, 21, 10) AS NCHAR(5)) AS CustomerID
FROM (SELECT EmployeeID,
        MAX(CAST(OrderDate         AS BINARY(8))
              + CAST(RequiredDate  AS BINARY(8))
              + CAST(OrderID AS BINARY(4))
              + CAST(CustomerID    AS BINARY(10))
              ) AS binstr
      FROM dbo.Orders
      GROUP BY EmployeeID) AS D;
```

פונקציות צבירה רצות

פונקציות צבירה רצות הן צבירות של נתונים על פני רצף (לרוב מבוסס–זמן). יש וריאציות רבות של בעיות פונקציות צבירה רצות, ואתאר כאן כמה חשובות.

בדוגמאות הבאות אשתמש בטבלת סיכום הנקראת EmpOrders אשר מכילה שורה אחת לכל עובד וחודש, עם כמות ההזמנות הכוללת שבוצעו על ידי עובד זה בחודש זה. הרץ את הקוד בקטע–קוד 1–6 כדי ליצור את הטבלה EmpOrders ולמלא אותה בנתונים לדוגמה.

קטע–קוד 1–6: יצירה ומילוי של טבלת EmpOrders

```
USE tempdb;
GO

IF OBJECT_ID('dbo.EmpOrders') IS NOT NULL
  DROP TABLE dbo.EmpOrders;
GO

CREATE TABLE dbo.EmpOrders
(
  empid    INT       NOT NULL,
  ordmonth DATETIME  NOT NULL,
  qty      INT       NOT NULL,
  PRIMARY KEY(empid, ordmonth)
);

INSERT INTO dbo.EmpOrders(empid, ordmonth, qty)
  SELECT O.EmployeeID,
    CAST(CONVERT(CHAR(6), O.OrderDate, 112) + '01'
      AS DATETIME) AS ordmonth,
    SUM(Quantity) AS qty
  FROM Northwind.dbo.Orders AS O
    JOIN Northwind.dbo.[Order Details] AS OD
      ON O.OrderID = OD.OrderID
  GROUP BY EmployeeID,
    CAST(CONVERT(CHAR(6), O.OrderDate, 112) + '01'
      AS DATETIME);
```

טיפ: כל חודש יוצג על ידי תאריך ההתחלה שלו ויאוחסן כ–DATETIME. דבר זה יאפשר מניפולציה גמישה של הנתונים, תוך שימוש בפונקציות הקשורות לתאריך. כדי להבטיח שהערך יהיה תקין בטיפוס הנתונים, אחסנתי את היום הראשון של החודש לייצוג של "יום". כמובן שאתעלם מכך בחישובים שלי.

הרץ את השאילתה הבאה לקבלת התוכן של טבלת EmpOrders, המוצג בצורה מקוצרת בטבלה 3–6:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty
FROM dbo.EmpOrders
ORDER BY empid, ordmonth;
```

טבלה 3–6: תוכן טבלת EmpOrders (מקוצר)

empid	*ordmonth*	*qty*
1	1996-07	121
1	1996-08	247
1	1996-09	255
1	1996-10	143
1	1996-11	318
1	1996-12	536
1	1997-01	304
1	1997-02	168
1	1997-03	275
1	1997-04	20
...	...	...
2	1996-07	50
2	1996-08	94
2	1996-09	137
2	1996-10	248
2	1996-11	237
2	1996-12	319
2	1997-01	230
2	1997-02	36
2	1997-03	151
2	1997-04	468
...	...	...

בהמשך אדון בשלושה סוגים של בעיות פונקציות רצות: מצטברות, נעות ומתחילת–תקופה (YTD).

פונקציות צבירה מצטברות

פונקציות צבירה מצטברות צוברות נתונים מהאלמנט הראשון ברצף עד לנקודה הנוכחית. למשל, תאר לך את הבקשה הבאה: לכל עובד וחודש, החזר את הכמות הכוללת והכמות החודשית הממוצעת, מתחילת פעילותו של העובד ועד לחודש המבוקש.

זכור את השיטות מבוססות–הסטים טרם SQL Server 2005 לחישוב מספרי שורה; בעזרת שימוש בשיטות אלו, אתה סורק את אותן שורות שעלינו לסרוק כעת לחישוב הכמויות הכוללות. ההבדל הוא שעבור מספרי שורה השתמשת בפונקציית צבירה COUNT, ואילו כאן אתה מתבקש לספק SUM ו–AVG. הדגמתי שני פתרונות לחישוב מספרי שורה – אחד על ידי שימוש בתת–שאילתות ואחד על ידי שימוש ב–joins. בפתרון שמשתמש ב–joins, יישמתי את מה שאני מכנה **שיטת פרישה–קריסה (expand-collapse technique)**. עבורי, הפתרון של תת–השאילתות הוא אינטואיטיבי הרבה יותר מאשר פתרון ה–join, עם שיטת הפרישה–קריסה המלאכותית שלו. כך שכאשר אין הבדלי ביצועים, אעדיף להשתמש בתת–שאילתות. בדרך כלל, לא תראה הבדלי ביצועים כאשר מעורבת פונקציית צבירה אחת בלבד, שכן התוכניות תהיינה דומות. אך כאשר אתה מבקש פונקציות צבירה מרובות, פתרון תת–השאילתה עשוי להביא לתוכנית הסורקת את הנתונים בנפרד לכל פונקציית צבירה. השווה זאת לתוכנית של פתרון ה–join, אשר לרוב מחשבת את כל פונקציות הצבירה תוך כדי סריקה יחידה של נתוני המקור.

כך שהבחירה שלי בדרך כלל פשוטה – השתמש בתת–שאילתה עבור פונקציית צבירה אחת, ו–join עבור פונקציות צבירה מרובות. השאילתה הבאה מיישמת את גישת הפרישה–קריסה לייצור התוצאה הרצויה, המוצגת בצורה מקוצרת בטבלה 4–6:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

טבלה 4–6: פונקציות צבירה מצטברות לעובד, חודש (מקוצר)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
1	1996-12	536	1620	270.00
1	1997-01	304	1924	274.86
1	1997-02	168	2092	261.50
1	1997-03	275	2367	263.00
1	1997-04	20	2387	238.70
...	...	...	...	...
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	1996-12	319	1085	180.83
2	1997-01	230	1315	187.86
2	1997-02	36	1351	168.88
2	1997-03	151	1502	166.89
2	1997-04	468	1970	197.00
...	...	...	...	...

כעת הבה נניח שהתבקשת להחזיר רק פונקציית צבירה אחת (נאמר, כמות כוללת). אתה יכול להשתמש בבטחה בגישת תת–השאילתה:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  (SELECT SUM(O2.qty)
   FROM dbo.EmpOrders AS O2
   WHERE O2.empid = O1.empid
     AND O2.ordmonth <= O1.ordmonth) AS totalqty
FROM dbo.EmpOrders AS O1
GROUP BY O1.empid, O1.ordmonth, O1.qty;
```

שים לב: בשני המקרים, אותן סוגיות ביצועים של N^2 בהן דנתי בנוגע למספרי שורה קיימות גם כאן. מכיוון שפונקציות צבירה רצות לרוב מחושבות על מספר קטן יחסית של שורות בכל קבוצה, לסוגיות הביצועים לא תהיה השפעה משמעותית, בהנחה שיש לך אינדקסים נאותים (grouping_columns, sort_columns, covering_columns).

ANSI SQL:2003 והרחבות OLAP ל–ANSI SQL:1999 מספקות תמיכה לפונקציות צבירה רצות על ידי פונקציות צבירה חלוניות. כפי שהזכרתי קודם, SQL Server 2005 יישם את הפסוקית OVER לפונקציות צבירה רק עם הפסוקית PARTITION BY. לפי ANSI באפשרותך לספק פתרון תוך הישענות אך ורק על פונקציות חלוניות כלהלן:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty,
  SUM(O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth) AS totalqty,
  CAST(AVG(1.*O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth)
  AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders;
```

כאשר קוד זה ייתמך לבסוף ב– SQL Server, תוכל לצפות לשיפורי ביצועים משמעותיים, וכמובן לשאילתות פשוטות הרבה יותר.

ייתכן שגם תתבקש לסנן את הנתונים – למשל, החזרת פונקציות צבירה חודשיות לכל עובד רק עבור החודשים בהם טרם השיג העובד יעד מסוים. לרוב, יהיה ברשותך יעד לכל עובד מאוחסן בטבלת יעדים אותה תצטרך לצרף. כדי לפשט את הדוגמה, אניח שלכל העובדים יעד זהה של כמות כוללת – 1000. בפועל, תשתמש במאפיין יעד מטבלת היעדים. מכיוון שעליך לסנן פונקציית צבירה, ולא מאפיין, עליך לציין את הביטוי המסנן (במקרה זה, SUM(O2.qty) < 1000) בפסוקית HAVING, ולא בפסוקית WHERE. הפתרון מוצג להלן והוא יפיק את הפלט המוצג בצורה מקוצרת בטבלה 5–6:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) < 1000
ORDER BY O1.empid, O1.ordmonth;
```

טבלה 5–6: פונקציות צבירה מצטברות, כאשר totalqty < 1000

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
3	...	...	...	...

הדברים מסתבכים מעט אם עליך גם לכלול את השורות עבור חודשים בהם העובדים השיגו את היעד שלהם. אם תציין SUM(O2.qty) <= 1000 (כלומר, תרשום => במקום <), עדיין לא תקבל את השורה בה העובד השיג את היעד שלו, אלא אם כן הכמות המצטברת עד חודש זה וכולל, היא בדיוק 1000. אך זכור שיש לך גישה הן לסך הכמות המצטברת והן לכמות בחודש הנוכחי, ושימוש בשני ערכים אלו ביחד יאפשר לך לפתור את הבעיה. אם תשנה את המסנן HAVING ל: SUM(O2.qty) – O1.qty < 1000 תקבל את החודשים בהם הכמות הכוללת של העובד, **ללא ההזמנות של החודש הנוכחי**, נמוכה מהיעד. בפרט, החודש הראשון בו עובד השיג או עבר את היעד מספק את הקריטריון החדש הזה, וחודש זה יופיע בתוצאות. להלן הפתרון המלא המפיק את הפלט המוצג בצורה מקוצרת בטבלה 6–6:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) - O1.qty < 1000
ORDER BY O1.empid, O1.ordmonth;
```

טבלה 6–6: פונקציות צבירה מצטברות, עד ש–totalqty מגיע לראשונה ל–1000 או יותר (מקוצר)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	1996-12	319	1085	180.83
3	...	...	...	...

שים לב: ייתכן שתחשוב על פתרון אחר שייראה כחלופה אפשרית ופשוטה יותר – להניח לתנאי ה–SUM אך לשנות את תנאי ה–join ל– O2.ordmonth < O1.ordmonth. בדרך זו, השאילתה תבחר שורות בהן הכמות המצטברת עד החודש הקודם (כולל), לא עמדה ביעד. אף על פי כן, בסופו של דבר, הפתרון אינו קל יותר (למשל, את AVG קשה לייצר); וגרוע מכך, אתה עלול להציע פתרון שאינו עובד עבור עובדים המגיעים ליעד בחודש הראשון שלהם.

נניח שהיית מעוניין לראות את התוצאות רק עבור החודש המסוים שבו העובד הגיע ליעד 1000, ולא את התוצאות של החודשים הקודמים. מה נכון רק לשורות אלו בטבלה 6–6? מה שאתה מחפש הן שורות מטבלה 6–6 בהן הכמות הכוללת גדולה מ– או שווה ל–1000. פשוט הוסף קריטריון זה למסנן HAVING. להלן השאילתה, שתייצר את הפלט המוצג בטבלה 6–7:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
```

```
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) - O1.qty < 1000
  AND SUM(O2.qty) >= 1000
ORDER BY O1.empid, O1.ordmonth;
```

טבלה 7–6: פונקציות צבירה מצטברות רק עבור חודשים בהם totalqty הגיעה לראשונה ליעד או עברה אותו

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-11	318	1084	216.80
2	1996-12	319	1085	180.83
3	1997-01	364	1304	186.29
4	1996-10	613	1439	359.75
5	1997-05	247	1213	173.29
6	1997-01	64	1027	171.17
7	1997-03	191	1069	152.71
8	1997-01	305	1228	175.43
9	1997-06	161	1007	125.88

פונקציות צבירה נעות

פונקציות צבירה נעות מחושבות על פני חלון נע בתוך רצף (שוב, בדרך כלל של זמן), בניגוד לחישוב מתחילת הרצף ועד לנקודה הנוכחית. **ממוצע נע** – כמו הכמות הממוצעת של העובד בשלושת החודשים האחרונים – היא דוגמה אחת לפונקציית צבירה נעה.

שים לב: ללא הבהרה, ביטויים כמו "שלושת החודשים האחרונים" יכולים להשתמע לשתי פנים. הכוונה בשלושת החודשים האחרונים יכולה להיות שלושת החודשים הקודמים (**לא כולל חודש זה**), או שהיא יכולה להיות שני החודשים הקודמים **וגם חודש זה**. כשאתה מקבל בעיה מעין זו, ודא שאתה יודע בדיוק באיזה חלון זמן אתה משתמש לפונקציית צבירה – לשורה מסוימת, היכן בדיוק מתחיל ונגמר החלון?

בדוגמה שלנו, חלון הזמן גדול מהנקודה בזמן שמתחילה לפני שלושה חודשים וקטן מ– או שווה לנקודה הנוכחית בזמן. שים לב שהגדרה זו תהיה טובה אפילו במקרים בהם אתה אחרי יחידות זמן גרעיניות יותר מאשר חודש (כולל יום, שעה, דקה, שנייה ומאית שנייה). הגדרה זו מטפלת גם בסוגיות המרה עקיפות

הנגרמות עקב רמת הדיוק הנתמכת על ידי SQL Server לטיפוס הנתונים DATETIME – 3.33 אלפיות שנייה. עדיף להשתמש בסימנים < ו– => מאשר בפסוקית BETWEEN כדי להימנע מסוגיות המרה עקיפות.

ההבדל העיקרי בין הפתרון לפונקציות צבירה מצטברות והפתרון לפונקציות צבירה נעות הוא בתנאי ה–join (או במסנן של תת–השאילתה, במקרה של הפתרון החלופי המשתמש בתת–שאילתות). במקום להשתמש ב– O2.ordmonth <= O1.current_month, השתמש ב– O2.ordmonth > three_months_before_current AND O2.ordmonth <= current_month. ב–T-SQL, נוסחה זו מיתרגמת לשאילתה הבאה, המפיקה את הפלט המופיע בצורה מקוצרת בטבלה 8–6:

```
SELECT O1.empid,
  CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND (O2.ordmonth > DATEADD(month, -3, O1.ordmonth)
     AND O2.ordmonth <=  O1.ordmonth)
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

טבלה 8–6: פונקציות צבירה נעות לעובד בשלושת החודשים עד הנוכחי (מקוצר)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	645	215.00
1	1996-11	318	716	238.67
1	1996-12	536	997	332.33
1	1997-01	304	1158	386.00
1	1997-02	168	1008	336.00
1	1997-03	275	747	249.00
1	1997-04	20	463	154.33

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
...	...	...	...	...
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	479	159.67
2	1996-11	237	622	207.33
2	1996-12	319	804	268.00
2	1997-01	230	786	262.00
2	1997-02	36	585	195.00
2	1997-03	151	417	139.00
2	1997-04	468	655	218.33
...	...	...	...	...

שים לב שפתרון זה כולל פונקציות צבירה לתקופות של שלושה חודשים שאינם מכילים שלושה חודשים של נתונים בפועל. אם ברצונך להחזיר רק תקופות עם שלושה חודשים מלאים מצטברים, ללא שתי התקופות הראשונות שאינן מכסות שלושה חודשים, תוכל להוסיף למסנן HAVING את הקריטריון MIN(O2.ordmonth) = DATEADD(month, -2, O1.ordmonth).

שים לב: בנוסף לתמיכה באלמנטים הן של PARTITION BY והן של ORDER BY בפסוקית OVER לפונקציות צבירה מבוססות–חלון, ANSI תומכת גם בפסוקית ROWS המאפשרת לך לבקש פונקציות צבירה נעות. למשל, להלן השאילתה שתחזיר את התוצאה הרצויה לבקשה האחרונה לפונקציות צבירה נעות (בהנחה שבנתונים יש בדיוק שורה אחת לכל חודש):

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth,
  qty AS qtythismonth,
  SUM(O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth
                   ROWS 2 PRECEDING) AS totalqty,
  CAST(AVG(1.*O2.qty) OVER(PARTITION BY empid ORDER BY ordmonth
                           ROWS 2 PRECEDING)
    AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders;
```

מתחילת–תקופה (YTD – Year-To-Date)

פונקציות צבירה YTD צוברות ערכים מתחילת תקופה בהתבסס על יחידת DATETIME כלשהי (נאמר, שנה) עד לנקודה הנוכחית. החישוב דומה מאוד לפתרון של פונקציות צבירה נעות. ההבדל היחיד הוא הגבול הנמוך המסופק על ידי המסנן של השאילתה, שהוא חישוב תחילת השנה. למשל, השאילתה הבאה מחזירה פונקציות צבירה YTD לכל עובד וחודש, ומפיקה את הפלט המוצג בצורה מקוצרת בטבלה 9–6:

```
SELECT O1.empid,
  CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS DECIMAL(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND (O2.ordmonth >= CAST(CAST(YEAR(O1.ordmonth) AS CHAR(4))
                              + '0101' AS DATETIME)
         AND O2.ordmonth <= O1.ordmonth)
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

טבלה 9–6: פונקציות צבירה YTD לעובד, חודש (מקוצר)

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
1	1996-12	536	1620	270.00
1	1997-01	304	304	304.00
1	1997-02	168	472	236.00
1	1997-03	275	747	249.00
1	1997-04	20	767	191.75
...	...	...	...	...

empid	*ordmonth*	*qtythismonth*	*totalqty*	*avgqty*
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	1996-12	319	1085	180.83
2	1997-01	230	230	230.00
2	1997-02	36	266	133.00
2	1997-03	151	417	139.00
2	1997-04	468	885	221.25
...	...	...	...	...

סיבוב על ציר – משורות לטורים (Pivoting)

Pivoting היא שיטה המאפשרת לך לסובב שורות לטורים, עם אפשרות של ביצוע פונקציות צבירה כחלק מהתהליך. מספר היישומים לסיבוב על ציר הוא עצום. בסעיף זה אציג כמה מהם, ביניהם מאפייני סיבוב בסביבה של סכמה פתוחה (Open Schema), פתרון בעיות של חלוקה רלציונית, ועיצוב נתוני צבירה (אגרגציה). בהמשך הפרק וכן בפרקים אחרים בספר, אציג יישומים נוספים. כרגיל בספר זה, אציג פתרונות המתאימים לגרסאות קודמות ל– SQL Server 2005 וכן פתרונות המשתמשים באופרטורים חדשים ולפיכך ניתנים ליישום רק ב– SQL Server 2005.

סיבוב מאפיינים

כתרחיש לסיבוב מאפיינים (pivoting attributes) אשתמש **בסכמה פתוחה** (Open Schema). סכמה פתוחה היא מבנה סכמה שאתה יוצר כדי להתמודד עם שינויי סכמה תכופים. המודל הרלציוני ו–SQL עושים עבודה טובה מאוד במניפולציה של נתונים (DML), הכוללת שינוי נתונים ואחזורם. אך שפת הגדרת הנתונים (DDL) של SQL לא מקלה על ההתמודדות עם שינויי סכמה תכופים. בכל פעם שעליך להוסיף ישויות חדשות, עליך ליצור טבלאות חדשות; בכל פעם שישויות קיימות משנות את מבנן, עליך להוסיף, לשנות או להסיר טורים. שינויים כאלו גורמים בדרך כלל אי–זמינות של האובייקטים המושפעים, והם גורמים גם לשינויים משמעותיים ביישום.

בתרחיש של שינויי סכמה רבים, באפשרותך לאחסן את כל הנתונים בטבלה יחידה, שבה כל ערך מאפיין יושב בשורה נפרדת עם ה–ID של האובייקט ועם השם או ה–ID של המאפיין. לייצוג ערכי המאפיין אתה משתמש בטיפוס הנתונים SQL_VARIANT כדי לאחסן סוגי מאפיינים שונים בטור אחד.

בדוגמאות שלי, אשתמש בטבלה OpenSchema שבאפשרותך ליצור ולמלא על ידי הרצת הקוד בקטע–קוד 2–6.

קטע–קוד 2–6: יצירה ומילוי של טבלת OpenSchema

```
SET NOCOUNT ON;
USE tempdb;
GO

IF OBJECT_ID('dbo.OpenSchema') IS NOT NULL
  DROP TABLE dbo.OpenSchema;
GO

CREATE TABLE dbo.OpenSchema
(
  objectid  INT          NOT NULL,
  attribute NVARCHAR(30) NOT NULL,
  value     SQL_VARIANT  NOT NULL,
  PRIMARY KEY (objectid, attribute)
);

INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(1, N'attr1', CAST('ABC'      AS VARCHAR(10))  );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(1, N'attr2', CAST(10         AS INT)          );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(1, N'attr3', CAST('20040101' AS SMALLDATETIME));
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr2', CAST(12         AS INT)          );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr3', CAST('20060101' AS SMALLDATETIME));
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr4', CAST('Y'        AS CHAR(1))      );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(2, N'attr5', CAST(13.7       AS DECIMAL(9,3)) );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(3, N'attr1', CAST('XYZ'      AS VARCHAR(10))  );
```

```
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(3, N'attr2', CAST(20         AS INT)          );
INSERT INTO dbo.OpenSchema(objectid, attribute, value)
  VALUES(3, N'attr3', CAST('20050101' AS SMALLDATETIME));
```

התוכן של טבלת OpenSchema מוצג בטבלה 10–6.

טבלה 10–6: תוכן של טבלת OpenSchema

objectid	*attribute*	*value*
1	attr1	ABC
1	attr2	10
1	attr3	2004-01-01 00:00:00.000
2	attr2	12
2	attr3	2006-01-01 00:00:00.000
2	attr4	Y
2	attr5	13.700
3	attr1	XYZ
3	attr2	20
3	attr3	2005-01-01 00:00:00.000

ייצוג נתונים בצורה כזו מאפשר לשינויים לוגיים בסכמה להיות מיושמים ללא הוספה, שינוי או הסרה של טבלאות וטורים, אלא על ידי שימוש ב–INSERT, UPDATE ו–DELETE של DML במקום. כמובן שהיבטים אחרים של עבודה עם הנתונים (כמו אכיפת integrity, כוונון ואחזור) הופכים להיות יותר מורכבים ויקרים בייצוג כזה. קיימות גישות אחרות להתמודדות עם שינויים תכופים בהגדרת הנתונים – למשל, אחסנת הנתונים במבנה XML. עם זאת, כאשר אתה שוקל את היתרונות והחסרונות של כל ייצוג, ייתכן שתמצא את ייצוג הסכמה הפתוחה שהודגם כאן עדיף בתרחישים מסוימים – למשל, ייצוג נתוני מכירה פומבית.

זכור שייצוג זה של הנתונים דורש שאילתות מורכבות מאוד אפילו לבקשות פשוטות, מכיוון שמאפיינים שונים של אותו מופע ישות פרושים על פני מספר שורות. בטרם תבצע שאילתות על נתונים כאלו, ייתכן שתרצה לסובב אותם למבנה מסורתי עם טור אחד לכל מאפיין – אולי לאחסן את התוצאה בטבלה זמנית, להוסיף לה אינדקס, לבצע עליה שאילתות ואז להיפטר מהטבלה הזמנית. כדי לסובב את הנתונים ממבנה הסכמה הפתוחה שלהם למבנה מסורתי, עליך להשתמש בשיטת סיבוב על ציר.

בסעיף הבא אתאר את השלבים המעורבים בפתרון בעיות סיבוב על ציר. ברצוני לציין שכדי להבין את צעדי הפתרון, יעזור מאוד אם תחשוב על שלבי העיבוד הלוגי של שאילתה, אותם תיארתי בפרק 1. דנתי בשלבי עיבוד השאילתה המעורבים באופרטור הטבלאי PIVOT ב– SQL Server 2005, אך צעדים אלו קיימים גם בפתרון ב– SQL Server 2000. יתרה מכך, ב– SQL Server 2000 הצעדים מפורשים יותר בקוד, בעוד שב– SQL Server 2005 הם מרומזים.

הצעד הראשון בפתרון בעיות סיבוב על ציר הוא להבין כיצד מספר השורות בתוצאה מתקשר למספר השורות בנתוני המקור. כאן עליך ליצור שורת תוצאה יחידה מתוך מספר שורות בסיס לכל אובייקט. המשמעות של זה יכולה להיות ציון GROUP BY objectid.

כצעד הבא בבעיית סיבוב על ציר, אתה יכול לחשוב במושגים של טורי התוצאה. נדרש לך טור תוצאה לכל מאפיין ייחודי. מכיוון שהנתונים מכילים חמישה מאפיינים ייחודיים (attr1, attr2, attr3, attr4 ו–attr5), נדרשים לך חמישה ביטויים ברשימת ה–SELECT. כל ביטוי אמור לחלץ, מתוך השורות השייכות לאובייקט המקובץ, את הערך הקשור למאפיין מסוים. דבר זה יכול להיעשות בעזרת הביטוי MAX(CASE...) הבא, אשר בדוגמה זו מיושם על מאפיין attr2:

```
MAX(CASE WHEN attribute = 'attr2' THEN value END) AS attr2
```

זכור שללא פסוקית ELSE, CASE מניח ELSE NULL מרומז. הביטוי CASE המוצג לעיל ייצר NULL עבור שורות בהן attribute אינו שווה 'attr2' וייצר value כאשר attribute שווה 'attr2'. המשמעות היא שמבין השורות עם ערך נתון של objectid (נאמר, 1), הביטוי CASE ייצר מספר NULLs, ולכל היותר ערך אחד ידוע (10 בדוגמה שלנו), המייצג את הערך של מאפיין היעד (attr2 בדוגמה שלנו) עבור ה–objectid הנתון. הטריק לחילוץ הערך הידוע היחיד הוא להשתמש ב–MAX או ב–MIN. שניהם מתעלמים מ–NULLs ומחזירים את הערך הקיים היחיד שהוא לא NULL, מכיוון שהן המינימום והן המקסימום של סט המכיל ערך אחד הוא הערך עצמו. להלן השאילתה המלאה המסובבת על ציר את המאפיינים מטבלת OpenSchema, ומפיקה את הפלט המופיע בטבלה 11–6:

```
SELECT objectid,
  MAX(CASE WHEN attribute = 'attr1' THEN value END) AS attr1,
  MAX(CASE WHEN attribute = 'attr2' THEN value END) AS attr2,
  MAX(CASE WHEN attribute = 'attr3' THEN value END) AS attr3,
  MAX(CASE WHEN attribute = 'attr4' THEN value END) AS attr4,
  MAX(CASE WHEN attribute = 'attr5' THEN value END) AS attr5
FROM dbo.OpenSchema
GROUP BY objectid;
```

טבלה 11–6: OpenSchem מסובבת על ציר

objectid	*attr1*	*attr2*	*attr3*	*attr4*	*attr5*
1	ABC	10	2004-01-01 00:00:00.000	NULL	NULL
2	NULL	12	2006-01-01 00:00:00.000	Y	13.700
3	XYZ	20	2005-01-01 00:00:00.000	NULL	NULL

שים לב: כדי לכתוב שאילתה זו עליך לדעת את שמות המאפיינים. אם אינך יודע, יהיה עליך לבנות את מחרוזת השאילתה בצורה דינמית.

מידע נוסף: לפרטים על סיבוב דינמי על ציר, אנא פנה לספר:

Inside Microsoft SQL Server 2005: T-SQL Programming (Microsoft Press, 2006).

שיטה זו לסיבוב נתונים על ציר יעילה מאוד מכיוון שהיא סורקת את טבלת הבסיס פעם אחת בלבד.

SQL Server 2005 מציג PIVOT, אופרטור מובנה מיוחד לסיבוב על ציר. אני חייב לומר שאני מוצא אותו מאוד מבלבל ולא אינטואיטיבי. איני רואה יתרון רב בשימוש בו, מלבד העובדה שהוא מאפשר קוד קצר יותר. הוא אינו תומך בסיבוב דינמי על ציר, ומאחורי הקלעים הוא מפעיל לוגיקה דומה מאוד לזו שהצגתי בפתרון האחרון. כך שסביר להניח שלא תמצא אפילו הבדלים בולטים בביצועים. מכל מקום, להלן הדרך בה היית מסובב על ציר את נתוני OpenSchema תוך שימוש באופרטור PIVOT:

```
SELECT objectid, attr1, attr2, attr3, attr4, attr5
FROM dbo.OpenSchema
  PIVOT(MAX(value) FOR attribute
    IN([attr1],[attr2],[attr3],[attr4],[attr5])) AS P;
```

בפתרון זה תוכל לזהות את כל האלמנטים בהם השתמשתי בפתרון הקודם. הקלטים לאופרטור PIVOT הם כלהלן:

- פונקציית הצבירה שתופעל על הערכים בקבוצה. במקרה שלנו זו MAX(value), המחלצת את הערך היחיד שאינו NULL השייך למאפיין היעד. במקרים אחרים, ייתכן שיהיה לך יותר מערך אחד בקבוצה שאינו NULL ותרצה פונקציית צבירה שונה (למשל, SUM או AVG).
- לאחר מילת–המפתח FOR, טור המקור המחזיק את שמות טורי היעד (attribute, במקרה שלנו).
- רשימת שמות טורי היעד בפועל בסוגריים לאחר מילת–המפתח IN.

הקטע המבלבל פה הוא שאין שום פסוקית GROUP BY מפורשת, אלא שמתרחש קיבוץ מרומז. זה כאילו שפעילות הסיבוב על ציר מבוססת על קבוצות המוגדרות על ידי רשימת הטורים שלא הוזכרו בקלטים של PIVOT (בסוגריים שאחרי מילת–המפתח PIVOT). במקרה שלנו, objectid הוא הטור המגדיר את הקבוצות.

אזהרה: מכיוון שכל הטורים שלא צוינו מגדירים את הקבוצות, שלא במתכוון, אתה עלול לסיים עם קיבוץ לא רצוי. כדי לפתור זאת, השתמש בטבלה נגזרת או בביטוי טבלה שגור (CTE) המחזיר רק את הטורים המבוקשים, ומפעיל PIVOT לביטוי טבלה זה ולא לטבלת הבסיס. מייד אדגים זאת.

טיפ: הקלט לפונקציית הצבירה חייב להיות טור בסיס ללא מניפולציה – הוא אינו יכול להיות ביטוי (למשל: SUM(qty * price)). אם ברצונך לספק את פונקציית הצבירה עם ביטוי כקלט, צור טבלה נגזרת או CTE ותקצה לביטוי כינוי טור (qty * price AS value), ובשאילתה החיצונית השתמש בטור זה כקלט לפונקציית הצבירה של PIVOT ((SUM(value)).

כמו כן, אינך יכול לסובב על ציר מאפיינים המגיעים מיותר מאשר טור אחד (הטור שמופיע לאחר מילת–המפתח FOR). אם עליך לסובב על ציר מאפיינים של יותר מטור אחד (נאמר, empid ו– YEAR(orderdate)), באפשרותך להשתמש בגישה דומה להצעה הקודמת; צור טבלה נגזרת או CTE בהם אתה משרשר את הערכים מכל הטורים שברצונך לסובב והקצה לביטוי כינוי טור (CAST(empid AS VARCHAR(10)) + '_' + CAST(YEAR(orderdate) AS CHAR(4)) AS empyear). אז, בשאילתה החיצונית, ציין טור זה אחרי מילת–המפתח FOR של PIVOT (FOR empyear IN([1_2004], [1_2005], [1_2006], [2_2004], …)).

חלוקה רלציונית

סיבוב (pivoting) על ציר יכול גם לשמש לפתרון בעיות חלוקה רלציונית כאשר מספר האלמנטים בסט המחלק קטן יחסית. בדוגמאות שלי אשתמש בטבלת OrderDetails, אותה ניתן ליצור ולמלא בנתונים על ידי הרצת הקוד בקטע–קוד 3–6.

קטע–קוד 3–6: יצירה ומילוי של טבלת OrderDetails

```
USE tempdb;
GO

IF OBJECT_ID('dbo.OrderDetails') IS NOT NULL
  DROP TABLE dbo.OrderDetails;
GO
```

```
CREATE TABLE dbo.OrderDetails
(
  orderid   VARCHAR(10) NOT NULL,
  productid INT         NOT NULL,
  PRIMARY KEY(orderid, productid)
  /* other columns */
);

INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 1);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 2);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 3);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('A', 4);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('B', 2);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('B', 3);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('B', 4);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('C', 3);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('C', 4);
INSERT INTO dbo.OrderDetails(orderid, productid) VALUES('D', 4);
```

בעיה קלאסית של חלוקה רלציונית היא להחזיר הזמנות המכילות סל מסוים של מוצרים – נאמר, מוצרים 2, 3 ו–4. אתה משתמש בשיטה של סיבוב על ציר כדי לסובב רק את המוצרים הרלוונטיים לטורים נפרדים לכל הזמנה. במקום להחזיר ערך מאפיין, אתה מפיק 1 אם המוצר קיים בהזמנה ו–0 אחרת. אתה מייצר טבלה נגזרת משאילתת הסיבוב, ובשאילתה חיצונית מסנן רק הזמנות המכילות 1 בכל טורי המוצר. להלן השאילתה המלאה המחזירה את ההזמנות הנכונות A ו–B:

```
SELECT orderid
FROM (SELECT
        orderid,
        MAX(CASE WHEN productid = 2 THEN 1 END) AS P2,
        MAX(CASE WHEN productid = 3 THEN 1 END) AS P3,
        MAX(CASE WHEN productid = 4 THEN 1 END) AS P4
      FROM dbo.OrderDetails
      GROUP BY orderid) AS P
WHERE P2 = 1 AND P3 = 1 AND P4 = 1;
```

אם תריץ רק את השאילתה של הטבלה הנגזרת, תקבל את המוצרים המסובבים (pivoted products) לכל הזמנה כפי שמוצג בטבלה 12–6.

טבלה 12–6: תוכן טבלה נגזרת P

orderid	*P2*	*P3*	*P4*
A	1	1	1
B	1	1	1
C	NULL	1	1
D	NULL	NULL	1

כדי למלא את הבקשה בעזרת שימוש באופרטור PIVOT החדש, השתמש בשאילתה הבאה:

```
SELECT orderid
FROM (SELECT *
      FROM dbo.OrderDetails
        PIVOT(MAX(productid) FOR productid IN([2],[3],[4])) AS P) AS T
WHERE [2] = 2 AND [3] = 3 AND [4] = 4;
```

פונקציית הצבירה חייבת לקבל טור כקלט, כך שסיפקתי את ה–productid עצמו. המשמעות היא שאם המוצר קיים בהזמנה, הערך המתאים יכיל את ה–productid האמיתי ולא 1. זו הסיבה לכך שהמסנן נראה מעט שונה כאן.

שים לב שאתה יכול לגרום לשתי השאילתות להיות יותר אינטואיטיביות ודומות זו לזו בהיגיון שלהן, על ידי שימוש בפונקציית הצבירה COUNT במקום MAX. בדרך זו, שתי השאילתות יפיקו 1 כאשר המוצר קיים ו–0 כאשר אינו קיים (במקום NULL). להלן השאילתה כפי שתראה ב– SQL Server 2000:

```
SELECT orderid
FROM (SELECT
        orderid,
        COUNT(CASE WHEN productid = 2 THEN 1 END) AS P2,
        COUNT(CASE WHEN productid = 3 THEN 1 END) AS P3,
        COUNT(CASE WHEN productid = 4 THEN 1 END) AS P4
      FROM dbo.OrderDetails
      GROUP BY orderid) AS P
WHERE P2 = 1 AND P3 = 1 AND P4 = 1;
```

ולהלן השאילתה בה תשתמש ב– SQL Server 2005 :

```
SELECT orderid
FROM (SELECT *
      FROM dbo.OrderDetails
        PIVOT(COUNT(productid) FOR productid IN([2],[3],[4])) AS P) AS T
WHERE [2] = 1 AND [3] = 1 AND [4] = 1;
```

פונקציות צבירה של נתונים

ניתן להשתמש בשיטה של סיבוב על ציר לעיצוב נתוני פונקציות צבירה, לרוב למטרת הפקת דוחות. בדוגמאות שלי אשתמש בטבלה Orders, שניתן לייצר ולמלא על ידי הרצת הקוד בקטע-קוד 4-6.

קטע-קוד 4-6: יצירה ומילוי של טבלת Orders

```
USE tempdb;
GO

IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
GO

CREATE TABLE dbo.Orders
(
  orderid   int        NOT NULL PRIMARY KEY NONCLUSTERED,
  orderdate datetime   NOT NULL,
  empid     int        NOT NULL,
  custid    varchar(5) NOT NULL,
  qty       int        NOT NULL
);

CREATE UNIQUE CLUSTERED INDEX idx_orderdate_orderid
  ON dbo.Orders(orderdate, orderid);

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30001, '20020802', 3, 'A', 10);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(10001, '20021224', 1, 'A', 12);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(10005, '20021224', 1, 'B', 20);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(40001, '20030109', 4, 'A', 40);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(10006, '20030118', 1, 'C', 14);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(20001, '20030212', 2, 'B', 12);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(40005, '20040212', 4, 'A', 10);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(20002, '20040216', 2, 'C', 20);
```

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30003, '20040418', 3, 'B', 15);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30004, '20020418', 3, 'C', 22);
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
  VALUES(30007, '20020907', 3, 'D', 30);
```

התוכן של טבלת Orders מוצג בטבלה 13–6.

טבלה 13–6: תוכן טבלת Orders

orderid	*orderdate*	*empid*	*custid*	*qty*
30004	2002-04-18 00:00:00.000	3	C	22
30001	2002-08-02 00:00:00.000	3	A	10
30007	2002-09-07 00:00:00.000	3	D	30
10001	2002-12-24 00:00:00.000	1	A	12
10005	2002-12-24 00:00:00.000	1	B	20
40001	2003-01-09 00:00:00.000	4	A	40
10006	2003-01-18 00:00:00.000	1	C	14
20001	2003-02-12 00:00:00.000	2	B	12
40005	2004-02-12 00:00:00.000	4	A	10
20002	2004-02-16 00:00:00.000	2	C	20
30003	2004-04-18 00:00:00.000	3	B	15

נניח שברצונך להחזיר שורה לכל לקוח, עם כמויות שנתיות כוללות בטור נפרד לכל שנה. אתה משתמש בשיטת סיבוב על ציר דומה מאוד לקודמת שהצגתי, אלא שהפעם במקום להשתמש ב–MAX, אתה משתמש בפונקציית הצבירה SUM, אשר תחזיר את הפלט המוצג בטבלה 14–6:

```
SELECT custid,
  SUM(CASE WHEN orderyear = 2002 THEN qty END) AS [2002],
  SUM(CASE WHEN orderyear = 2003 THEN qty END) AS [2003],
  SUM(CASE WHEN orderyear = 2004 THEN qty END) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
GROUP BY custid;
```

טבלה 6–14: כמויות שנתיות כוללות לכל לקוח

custid	*2002*	*2003*	*2004*
A	22	40	10
B	20	12	15
C	22	14	20
D	30	NULL	NULL

כאן תוכל לראות את השימוש בטבלה נגזרת לצורך בידוד אך ורק של האלמנטים הרלוונטיים לפעילות הסיבוב על ציר (custid, orderyear, qty).

אחת הסוגיות העיקריות בפתרון זה לסיבוב על ציר היא, שאתה עלול לייצר מחרוזות שאילתה ארוכות כאשר מספר האלמנטים שעליך לסובב הוא גדול. בניסיון לקצר את מחרוזת השאילתה, תוכל להשתמש בטבלת מטריצה המכילה שורה וטור לכל מאפיין שעליך לסובב (orderyear במקרה זה). רק ערכי טור בהצטלבויות של שורות וטורים תואמים מכילים את הערך 1, ובשאר ערכי הטור יופיעו NULL או 0, בהתאם לצרכיך. הרץ את הקוד בקטע–קוד 6–5 ליצירה ומילוי של טבלת המטריצה.

קטע–קוד 6–5: יצירה ומילוי של טבלת המטריצה

```
USE tempdb;
GO

IF OBJECTPROPERTY(OBJECT_ID('dbo.Matrix'), 'IsUserTable') = 1
  DROP TABLE dbo.Matrix;
GO

CREATE TABLE dbo.Matrix
(
  orderyear INT NOT NULL PRIMARY KEY,
  y2002 INT NULL,
  y2003 INT NULL,
  y2004 INT NULL
);

INSERT INTO dbo.Matrix(orderyear, y2002) VALUES(2002, 1);
INSERT INTO dbo.Matrix(orderyear, y2003) VALUES(2003, 1);
INSERT INTO dbo.Matrix(orderyear, y2004) VALUES(2004, 1);
```

התוכן של טבלת המטריצה מוצג בטבלה 6–15.

טבלה 15–6: תוכן של טבלת מטריצה

orderyear	*y2002*	*y2003*	*y2004*
2002	1	NULL	NULL
2003	NULL	1	NULL
2004	NULL	NULL	1

אתה מאחד את טבלת הבסיס (או ביטוי טבלה) עם טבלת המטריצה בהתבסס על התאמה של orderyear. המשמעות היא שכל שורה מטבלת הבסיס תשויך לשורה אחת מהמטריצה – זו בעלת orderyear זהה. בשורה זו, רק ערכי טור תואמים של orderyear יכילו 1. כך תוכל להחליף את הביטוי

```
SUM(CASE WHEN orderyear = <some_year> THEN qty END) AS [<some_year>]
```

בביטוי השווה לו מבחינה לוגית

```
SUM(qty*y<some_year>) AS [<some_year>]
```

להלן השאילתה המלאה:

```
SELECT custid,
  SUM(qty*y2002) AS [2002],
  SUM(qty*y2003) AS [2003],
  SUM(qty*y2004) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  JOIN dbo.Matrix AS M ON D.orderyear = M.orderyear
GROUP BY custid;
```

אם אתה זקוק לפירוט מספר ההזמנות ולא מעוניין בסכום הכולל של qty, בפתרון המקורי אתה מפיק 1 במקום הטור qty לכל הזמנה, ומשתמש בפונקציית הצבירה COUNT, שתפיק את הפלט המוצג בטבלה 16–6:

```
SELECT custid,
  COUNT(CASE WHEN orderyear = 2002 THEN 1 END) AS [2002],
  COUNT(CASE WHEN orderyear = 2003 THEN 1 END) AS [2003],
  COUNT(CASE WHEN orderyear = 2004 THEN 1 END) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
GROUP BY custid;
```

טבלה 16–6: ספירה של כמות הזמנות שנתית לכל לקוח

custid	*2002*	*2003*	*2004*
A	2	1	1
B	1	1	1
C	1	1	1
D	1	0	0

עם מטריצת הטבלה, פשוט הגדר את הטור המתאים לשנת היעד:

```
SELECT custid,
  COUNT(y2002) AS [2002],
  COUNT(y2003) AS [2003],
  COUNT(y2004) AS [2004]
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
  JOIN dbo.Matrix AS M ON D.orderyear = M.orderyear
GROUP BY custid;
```

כמובן שכאשר משתמשים באופרטור PIVOT ב– SQL Server 2005, מחרוזות השאילתה מלכתחילה קצרות. להלן השאילתה המשתמשת באופרטור PIVOT לחישוב כמויות שנתיות כוללות ללקוח:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2002],[2003],[2004])) AS P;
```

ולהלן השאילתה שסופרת הזמנות:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
  PIVOT(COUNT(orderyear) FOR orderyear IN([2002],[2003],[2004])) AS P;
```

זכור ששאילתות סטטיות המבצעות סיבוב על ציר, דורשות ממך לדעת מראש את רשימת המאפיינים שבכוונתך לסובב. עבור סיבוב דינמי על ציר, עליך לבנות את מחרוזת השאילתה דינמית.

סיבוב על ציר – מטורים לשורות (Unpivoting)

unpivoting הוא ההיפך מ–pivoting – כלומר, סיבוב על ציר מטורים לשורות. unpivoting משמש בדרך כלל לנרמול נתונים, אך יש לו גם שימושים אחרים.

שים לב: Unpivoting אינו לחלוטין ההופכי של pivoting, שכן הוא לא בהכרח יאפשר לך לייצר מחדש שורות מקור שסובבו על ציר. עם זאת, לצורך הפשטות, חשוב על כך כפעולה ההפוכה ל–pivoting.

בדוגמאות שלי אשתמש בטבלה PvtCustOrders, אותה תייצר ותמלא על ידי הרצת הקוד בקטע–קוד 6–6.

קטע–קוד 6–6: יצירה ומילוי של טבלת PvtCustOrders

```
USE tempdb;
GO
IF OBJECT_ID('dbo.PvtCustOrders') IS NOT NULL
  DROP TABLE dbo.PvtCustOrders;
GO

SELECT *
INTO dbo.PvtCustOrders
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2002],[2003],[2004])) AS P;
```

טבלה 6–17: תוכן של טבלת PvtCustOrders

custid	*2002*	*2003*	*2004*
A	22	40	10
B	20	12	15
C	22	14	20
D	30	NULL	NULL

המטרה במקרה זה תהיה לייצר שורת תוצאה לכל לקוח ושנה, המכילה את קוד הלקוח (custid), שנת הזמנה (orderyear), וכמות (qty).

אתחיל בפתרון המתאים לגרסאות קודמות ל– SQL Server 2005. גם כאן, נסה לחשוב במונחים של עיבוד לוגי של שאילתה כפי שמתואר בפרק 1.

הצעד הראשון והחשוב ביותר בפתרון הוא לייצר שלושה עותקים של כל שורת בסיס – אחד לכל שנה. ניתן להשיג זאת על ידי ביצוע הצלבה בין טבלת הבסיס וטבלת עזר וירטואלית המכילה שורה אחת לכל שנה. רשימת ה–SELECT יכולה אז להחזיר את custid ואת orderyear, וכן לחשב את ה–qty של שנת היעד בעזרת ביטוי ה–CASE הבא:

```
CASE orderyear
  WHEN 2002 THEN [2002]
  WHEN 2003 THEN [2003]
  WHEN 2004 THEN [2004]
END AS qty
```

בדרך זו אתה משיג unpivoting, אך תקבל גם שורות המתאימות לערכי NULL בטבלת המקור (למשל, עבור לקוח D בשנים 2003 ו–2004). כדי להעלים שורות אלו, צור טבלה נגזרת משאילתת הפתרון, ובשאילתה החיצונית, העלם את השורות עם NULL בטור qty.

שים לב: בפועל, לרוב תאחסן 0 ולא NULL בתור הכמות ללקוח ללא הזמנות בשנה מסוימת; כמות ההזמנה ידועה כאפס, ולא כלא–ידוע. עם זאת, השתמשתי כאן ב–NULLs כדי להדגים את הטיפול ב–NULLs, שהוא צורך נפוץ בבעיות unpivoting.

להלן הפתרון המלא, המחזיר את הפלט הרצוי כפי שמוצג בטבלה 18–6:

```
SELECT custid, orderyear, qty
FROM (SELECT custid, orderyear,
        CASE orderyear
          WHEN 2002 THEN [2002]
          WHEN 2003 THEN [2003]
          WHEN 2004 THEN [2004]
        END AS qty
      FROM dbo.PvtCustOrders,
        (SELECT 2002 AS orderyear
         UNION ALL SELECT 2003
         UNION ALL SELECT 2004) AS OrderYears) AS D
WHERE qty IS NOT NULL;
```

טבלה 18–6: כמויות כוללות ללקוח ושנת הזמנה לאחר Unpivot

custid	*orderyear*	*qty*
A	2002	22
B	2002	20

custid	*orderyear*	*qty*
C	2002	22
D	2002	30
A	2003	40
B	2003	12
C	2003	14
A	2004	10
B	2004	15
C	2004	20

ב– SQL Server 2005 העניינים פשוטים יותר בצורה משמעותית. אתה משתמש באופרטור הטבלאי UNPIVOT כלהלן:

```
SELECT custid, orderyear, qty
FROM dbo.PvtCustOrders
  UNPIVOT(qty FOR orderyear IN([2002],[2003],[2004])) AS U
```

שלא כמו האופרטור PIVOT, אני מוצא את האופרטור UNPIVOT פשוט ואינטואיטיבי, וללא ספק דורש משמעותית פחות קוד. הקלט הראשון של UNPIVOT הוא שם טור היעד שיחזיק את ערכי המאפיין המסובב (qty). אז, לאחר מילת המפתח FOR, אתה מגדיר את שם טור היעד שיחזיק את שמות הטורים המסובבים (orderyear). לבסוף, בסוגריים של הפסוקית IN, אתה מגדיר את שמות טורי המקור שברצונך לסובב ([2004], [2003], [2002]).

טיפ: כל מאפייני הבסיס עליהם מבוצע unpivot חייבים להיות מאותו טיפוס נתונים. אם ברצונך לבצע unpivot על מאפיינים מטיפוסי נתונים שונים, צור טבלה נגזרת או CTE כאשר אתה ראשית ממיר את כל המאפיינים הרצויים ל–SQL_VARIANT. טור היעד אשר יחזיק ערכים עליהם בוצע unpivot יוגדר גם הוא כטיפוס SQL_VARIANT, ובתוך טור זה, הערכים יישמרו את טיפוס הנתונים המקורי שלהם.

שים לב: כמו PIVOT, UNPIVOT דורש רשימה ידועה מראש של שמות טור אותם יש לסובב.

פונקציות צבירה מוגדרות–משתמש

פונקציות צבירה (אגרגציות) מוגדרות–משתמש הן פונקציות צבירה שאינן מסופקות כפונקציות מובנות במוצר – למשל, שרשור מחרוזות, חישוב מכפלות, ביצוע מניפולציות מבוססות–סיבית, חישוב חציונים ואחרות. בקטע זה, אספק פתרונות למספר בקשות לפונקציות צבירה מותאמות. כמה מהשיטות בהן אשתמש הן כלליות – במובן שבאפשרותך להשתמש בהיגיון דומה לבקשות פונקציות צבירה אחרות – בעוד שאחרות הן ספציפיות לסוג מסוים של בקשת צבירה.

מידע נוסף: אחת משיטות פונקציות הצבירה מוגדרות-המשתמש הכלליות משתמשת בסמנים. לפרטים על סמנים, כולל טיפול בפונקציות צבירה מוגדרות-משתמש עם סמנים, אנא פנה לספר: Inside Microsoft SQL Server 2005: T-SQL Programming.

בדוגמאות שלי אשתמש בטבלה הכללית Groups, אותה אתה יוצר וממלא על ידי הרצת הקוד בקטע–קוד 7–6.

קטע–קוד 7–6: יצירה ומילוי של טבלת Groups

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Groups') IS NOT NULL
  DROP TABLE dbo.Groups;
GO

CREATE TABLE dbo.Groups
(
  groupid  VARCHAR(10) NOT NULL,
  memberid INT         NOT NULL,
  string   VARCHAR(10) NOT NULL,
  val      INT         NOT NULL,
  PRIMARY KEY (groupid, memberid)
);

INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 3, 'stra1', 6);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 9, 'stra2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 2, 'strb1', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 4, 'strb2', 7);
```

```
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 5, 'strb3', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 9, 'strb4', 11);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 3, 'strc1', 8);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 7, 'strc2', 10);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 9, 'strc3', 12);
```

התוכן של טבלת Groups מוצג בטבלה 19–6.

טבלה 19–6: תוכן טבלת Groups

groupid	*memberid*	*string*	*val*
a	3	stra1	6
a	9	stra2	7
b	2	strb1	3
b	4	strb2	7
b	5	strb3	3
b	9	strb4	11
c	3	strc1	8
c	7	strc2	10
c	9	strc3	12

לטבלת Groups יש טור המייצג את הקבוצה (groupid), טור המייצג מזהה ייחודי בתוך הקבוצה (memberid), וטורי ערכים (string ו–val) להם יש לבצע צבירה. אני אוהב להשתמש בטבלה כללית כזו של נתונים מכיוון שהדבר מאפשר לך להתמקד בשיטות ולא בנתונים. שים לב שזוהי פשוט צורה כללית של טבלה המכילה נתונים עליהם ברצונך לבצע צבירה. למשל, היא יכולה לייצג טבלת מכירות בה groupid מיוצג על ידי empid, val מיוצג על ידי qty וכך הלאה.

פונקציות צבירה מוגדרות–משתמש המשתמשות ב–Pivoting

שיטת מפתח אחת לפתרון בעיות פונקציות צבירה מוגדרות–משתמש היא pivoting. עקרונית אתה מסובב על ציר את הערכים שצריכים להשתתף בחישוב הצבירה; כאשר

כולם מופיעים באותה שורת תוצאה, אתה מבצע את החישוב בצורה ליניארית על פני הטורים. משתי סיבות, שיטת סיבוב על ציר זו מוגבלת למצבים בהם יש מספר קטן של אלמנטים בכל קבוצה. ראשית, עם מספר גדול של אלמנטים תגיע למחרוזות שאילתה ארוכות מאוד, דבר שאינו ריאלי. שנית, אלא אם כן יש לך טור רצף בתוך הקבוצה, תצטרך לחשב מספרי שורה שישמשו לזיהוי מיקום האלמנטים בתוך הקבוצה. למשל, אם עליך לשרשר את כל הערכים מהטור string לכל קבוצה, מה תגדיר כרשימת הערכים המסובבים? הערכים בטור memberid אינם ידועים מראש, וכן הם שונים בכל קבוצה. מספרי שורה המייצגים מיקומים בתוך הקבוצה פותרים את הבעיה. זכור שבגרסאות קודמות ל– SQL Server 2005, חישוב מספרי שורה יקר עבור קבוצות גדולות.

שרשור מחרוזות על ידי שימוש בסיבוב על ציר

כדוגמה ראשונה, השאילתה הבאה מחשבת צבירה של שרשור מחרוזת על פני הטור string לכל קבוצה עם שיטת סיבוב על ציר, המייצרת את הפלט המוצג בטבלה 20–6:

```
SELECT groupid,
    MAX(CASE WHEN rn = 1 THEN string ELSE '' END)
  + MAX(CASE WHEN rn = 2 THEN ',' + string ELSE '' END)
  + MAX(CASE WHEN rn = 3 THEN ',' + string ELSE '' END)
  + MAX(CASE WHEN rn = 4 THEN ',' + string ELSE '' END) AS string
FROM (SELECT groupid, string,
        (SELECT COUNT(*)
         FROM dbo.Groups AS B
         WHERE B.groupid = A.groupid
           AND B.memberid <= A.memberid) AS rn
      FROM dbo.Groups AS A) AS D
GROUP BY groupid;
```

טבלה 20–6: שרשור מחרוזות

groupid	*string*
a	stra1,stra2
b	strb1,strb2,strb3,strb4
c	strc1,strc2,strc3

השאילתה שמייצרת את הטבלה הנגזרת D מחשבת מספר שורה בתוך הקבוצה בהתבסס על מיון memberid. השאילתה החיצונית מסובבת על ציר את הערכים בהתבסס על מספרי השורה, ומבצעת שרשור ליניארי. אני מניח כאן שיש לכל היותר ארבע שורות בקבוצה, ולכן הגדרתי ארבעה ביטויי MAX(CASE…). נדרשים לך מספר ביטויי MAX(CASE…) כמספר המקסימלי של אלמנטים להם אתה צופה.

שים לב: חשוב להחזיר מחרוזת ריקה ולא NULL בפסוקית ELSE של הביטוי CASE. זכור ששרשור בין ערך ידוע ל–NULL מפיק NULL.

פונקציית הצבירה הכפלה על ידי שימוש בסיבוב על ציר

באופן דומה, ניתן לחשב את המכפלה של הערכים בטור val לכל קבוצה, ולהפיק את הפלט המוצג בטבלה 21–6:

```
SELECT groupid,
     MAX(CASE WHEN rn = 1 THEN val ELSE 1 END)
  * MAX(CASE WHEN rn = 2 THEN val ELSE 1 END)
  * MAX(CASE WHEN rn = 3 THEN val ELSE 1 END)
  * MAX(CASE WHEN rn = 4 THEN val ELSE 1 END) AS product
FROM (SELECT groupid, val,
        (SELECT COUNT(*)
         FROM dbo.Groups AS B
         WHERE B.groupid = A.groupid
           AND B.memberid <= A.memberid) AS rn
      FROM dbo.Groups AS A) AS D
GROUP BY groupid;
```

טבלה 21–6: פונקציית הצבירה הכפלה

groupid	*product*
a	42
b	693
c	960

הצורך בפונקציית הצבירה הכפלה נפוץ ביישומים כלכליים – למשל, לחישוב שיעורי ריבית דריבית.

פונקציות צבירה מוגדרות משתמש (UDA)

SQL Server 2005 מציג את היכולת לייצר פונקציות צבירה מוגדרות משתמש (UDA). כתיבת פונקציות צבירה מסוג זה (UDA) מתבצעת באחת משפות ה–NET. (למשל C# או NET. Microsoft Visual Basic), ואתה משתמש בהם ב–T-SQL. ספר זה מוקדש ל–T-SQL ולא ל– CLR (Common Language Runtime), כך שהוא לא יעסוק בדיונים ארוכים המסבירים פונקציות צבירה מוגדרות משתמש (UDA) של CLR. עם זאת, הספר יספק לך שתי דוגמאות עם הסברי צעד–אחר–צעד וכמובן, את ממשק ה–T-SQL המעורב. דוגמאות יסופקו הן ב–C# והן ב– Visual Basic.

קוד CLR בבסיס נתונים

סעיף זה דן בשילוב של .NET (Common Language Runtime) CLR ב– SQL Server 2005 ; לפיכך, ראוי להקדיש מספר מילים להסביר את ההיגיון שמאחורי שילוב CLR במסד הנתונים. חשוב גם לזהות את התרחישים בהם שימוש באובייקטים של CLR מתאים יותר מאשר שימוש ב–T-SQL.

פיתוח בשפות .NET. כמו C# ו–.NET. Visual Basic מספק מודל פיתוח עשיר ביותר. מסגרת העבודה של .NET. כוללת אלפי מחלקות מוכנות, ושימוש נבון בהן תלוי בך. שפות .NET. אינן רק מוכוונות–נתונים כמו SQL, ולכן אינך מוגבל כל–כך. למשל, regular expressions יעילים ביותר לבדיקת נכונות הנתונים, ונתמכים לחלוטין ב–.NET. שפות SQL הן מוכוונות–סטים ומבצעות באיטיות פעולות מוכוונות–שורה (שורה–אחר–שורה או שורה–אחת–בכל–פעם). לעיתים אתה זקוק לפעולות מוכוונות–שורה בתוך מסד הנתונים; מעבר מסמנים לקוד CLR אמור לשפר את הביצועים. יתרון נוסף של קוד CLR הוא שהוא יכול להיות מהיר הרבה יותר מאשר קוד T-SQL בפעולות חישוביות מורכבות.

למרות ש– SQL Server תמך בהרחבות תכנות גם לפני שהוצג שילוב של CLR, הרי ששילוב של CLR עולה על הרחבות אלו בכמה אופנים.

למשל, תוכל להוסיף שימושיות לגרסאות קודמות של SQL Server על ידי שימוש בפרוצדורות שמורות מורחבות (extended procedures). עם זאת, פרוצדורות כאלו יכולות לסכן את השלמות של העיבודים ב– SQL Server מכיוון שהזיכרון וניהול ה–threads שלהם אינם משולבים מספיק טוב עם ניהול המשאבים של SQL Server. קוד .NET. מנוהל על ידי CLR בתוך SQL Server, ומכיוון שה–CLR עצמו מנוהל על ידי SQL Server, הוא בטוח לשימוש הרבה יותר מאשר קוד פרוצדורות מורחבות.

T-SQL – שפה מוכוונת–סטים – עוצבה בעיקר כדי להתמודד עם נתונים והיא אופטימלית לצורך מניפולציה של נתונים. אינך צריך למהר לתרגם את כל קוד ה–T-SQL שלך לקוד CLR. שפת T-SQL היא עדיין השפה העיקרית של SQL Server. גישה לנתונים יכולה להתבצע רק דרך T-SQL. אם פעולה יכולה להיות מבוטאת כפעולה מוכוונת–סטים, עליך לתכנת אותה ב–T-SQL.

ישנה החלטה חשובה נוספת אליה עליך להגיע בטרם אתה מתחיל להשתמש בקוד CLR בתוך SQL Server. עליך להחליט היכן ירוץ קוד ה–CLR שלך – בשרת או בלקוח. קוד CLR הוא לרוב מהיר וגמיש יותר מאשר T-SQL לצורך חישובים, ולכן הוא מגדיל את ההזדמנויות לחישובים בצד השרת. עם זאת, צד השרת הוא לרוב קופסת עבודה אחת, ואיזון עומסים בשכבת הנתונים נמצא עדיין בשלבי ינקות. לפיכך, עליך לשקול אם יהיה זה נבון יותר לעבד חישובים אלו בצד הלקוח.

עם קוד CLR אתה יכול לכתוב פרוצדורות שמורות, טריגרים, פונקציות מוגדרות–משתמש, טיפוסים מוגדרי–משתמש ופונקציות צבירה מוגדרות–משתמש. שני האובייקטים האחרונים אינם יכולים להיכתב עם T-SQL; הם יכולים להיכתב רק בקוד CLR.

טיפוס מוגדר–משתמש (UDT) הוא סוג אובייקט ה–CLR המורכב ביותר ודורש כיסוי נרחב.

מידע נוסף: לפרטים אודות תכנות CLR UDTs וכן אודות תכנות רוטינות CLR, אנא פנה לספר Inside Microsoft SQL Server 2005: T-SQL Programming.

הבה נתחיל ביישום מעשי של שתי פונקציות UDA. הצעדים המעורבים ביצירת פונקציית UDA מבוססת–CLR הם כלהלן:

- הגדר את פונקציית ה–UDA כמחלקה בשפת NET..
- הדר (compile) את המחלקה שהגדרת לבניית מערך CLR.
- רשום את ה–assembly ב– SQL Server על ידי שימוש בפקודה CREATE ASSEMBLY.
- השתמש בפקודה CREATE AGGREGATE ב–T-SQL ליצירת פונקציית UDA הפונה ל–assembly הרשום.

שים לב: באפשרותך לרשום assembly וליצור אובייקט CLR ישירות מ– Microsoft Visual Studio 2005, תוך שימוש באפשרות של הטמעת פרויקט (פריט בתפריט Build>Deploy). סעיף זה יראה לך כיצד להטמיע אובייקטים של CLR ישירות מ– Visual Studio. כמו כן הייה מודע לכך שהטמעה ישירה מ– Visual Studio נתמכת רק במהדורת Professional או גבוהה יותר; אם אתה משתמש במהדורת Standard, האפשרות היחידה שלך היא הטמעה מפורשת ב– SQL Server.

סעיף זה יספק דוגמאות ליצירת פונקציית צבירה של שרשור מחרוזת וצבירה של פונקציות הכפלה הן ב–#C והן ב–Visual Basic .NET. תוכל למצוא את הקוד למחלקות #C בקטע–קוד 8–6 ואת הקוד למחלקות Visual Basic .NET בקטע–קוד 9–6. הדרישות לפונקציות צבירה מוגדרות משתמש (UDA) של CLR יסופקו לך במקביל לפיתוח של פונקציית UDA.

קטע–קוד 8–6: קוד UDA ב–#C

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
```

```
using System.Text;
using System.IO;
using System.Runtime.InteropServices;

[Serializable]
[SqlUserDefinedAggregate(
   Format.UserDefined,                // use user-defined serialization
   IsInvariantToDuplicates = false,  // duplicates make difference
    // for the result
   IsInvariantToNulls = true,        // don't care about NULLs
   IsInvariantToOrder = false,       // whether order makes difference
   IsNullIfEmpty = false,            // do not yield a NULL
    // for a set of zero strings
   MaxByteSize = 8000)]              // maximum size in bytes of
persisted value public struct CSStrAgg : IBinarySerialize
{
    private StringBuilder sb;
    private bool firstConcat;

    public void Init()
    {
        this.sb = new StringBuilder();
        this.firstConcat = true;
    }

    public void Accumulate(SqlString s)
    {
        if (s.IsNull)
        {
            return;                  // simply skip Nulls approach
        }
        if (this.firstConcat)
        {
            this.sb.Append(s.Value);
            this.firstConcat = false;
        }
        else
        {
            this.sb.Append(",");
            this.sb.Append(s.Value);
        }
    }
```

```
    public void Merge(CSStrAgg Group)
    {
        this.sb.Append(Group.sb);
    }

    public SqlString Terminate()
    {
        return new SqlString(this.sb.ToString());
    }

    public void Read(BinaryReader r)
    {
        sb = new StringBuilder(r.ReadString());
    }

    public void Write(BinaryWriter w)
    {
        if (this.sb.Length > 4000)   // check we don't
                                     // go over 8000 bytes

                                     // simply return first 8000 bytes
            w.Write(this.sb.ToString().Substring(0, 4000));
        else
            w.Write(this.sb.ToString());
    }

}   // end CSStrAgg

[Serializable]
[StructLayout(LayoutKind.Sequential)]
[SqlUserDefinedAggregate(
   Format.Native,                    // use native serialization
   IsInvariantToDuplicates = false, // duplicates make difference
    // for the result
   IsInvariantToNulls = true,        // don't care about NULLs
   IsInvariantToOrder = false)]      // whether order makes difference
public class CSProdAgg
{
    private SqlInt64 si;
    public void Init()
    {
        si = 1;
```

```
    }

    public void Accumulate(SqlInt64 v)
    {
        if (v.IsNull || si.IsNull)  // Null input = Null output approach
        {
            si = SqlInt64.Null;
            return;
        }
        if (v == 0 || si == 0)      // to prevent an exception in next if
        {
            si = 0;
            return;
        }
                                    // stop before we reach max value
        if (Math.Abs(v.Value) <= SqlInt64.MaxValue / Math.Abs(si.Value))
        {
            si = si * v;
        }
        else
        {
            si = 0;                 // if we reach too big value, return 0
        }

    }

    public void Merge(CSProdAgg Group)
    {
        Accumulate(Group.Terminate());
    }

    public SqlInt64 Terminate()
    {
        return (si);
    }

}  // end CSProdAgg
```

קטע-קוד 6-9: קוד UDA ב- Visual Basic .NET

```
Imports System
Imports System.Data
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Text
Imports System.IO
Imports System.Runtime.InteropServices


<Serializable(), _
 SqlUserDefinedAggregate( _
              Format.UserDefined, _
              IsInvariantToDuplicates:=True, _
              IsInvariantToNulls:=True, _
              IsInvariantToOrder:=False, _
              IsNullIfEmpty:=False, _
              MaxByteSize:=8000)> _
Public Class VBStrAgg
    Implements IBinarySerialize

    Private sb As StringBuilder
    Private firstConcat As Boolean = True

    Public Sub Init()
        Me.sb = New StringBuilder()
        Me.firstConcat = True
    End Sub


    Public Sub Accumulate(ByVal s As SqlString)
        If s.IsNull Then
            Return
        End If
        If Me.firstConcat = True Then
            Me.sb.Append(s.Value)
            Me.firstConcat = False
        Else
            Me.sb.Append(“,”)
            Me.sb.Append(s.Value)
        End If
    End Sub
```

```
    Public Sub Merge(ByVal Group As VBStrAgg)
        Me.sb.Append(Group.sb)
    End Sub

    Public Function Terminate() As SqlString
        Return New SqlString(sb.ToString())
    End Function

    Public Sub Read(ByVal r As BinaryReader) _
      Implements IBinarySerialize.Read
        sb = New StringBuilder(r.ReadString())
    End Sub

    Public Sub Write(ByVal w As BinaryWriter) _
      Implements IBinarySerialize.Write
        If Me.sb.Length > 4000 Then
            w.Write(Me.sb.ToString().Substring(0, 4000))
        Else
            w.Write(Me.sb.ToString())
        End If
    End Sub

End Class


<Serializable(), _
 StructLayout(LayoutKind.Sequential), _
 SqlUserDefinedAggregate( _
              Format.Native, _
              IsInvariantToOrder:=False, _
              IsInvariantToNulls:=True, _
              IsInvariantToDuplicates:=True)> _
Public Class VBProdAgg

    Private si As SqlInt64

    Public Sub Init()
        si = 1
    End Sub
```

```
    Public Sub Accumulate(ByVal v As SqlInt64)
        If v.IsNull = True Or si.IsNull = True Then
            si = SqlInt64.Null
            Return
        End If
        If v = 0 Or si = 0 Then
            si = 0
            Return
        End If
        If (Math.Abs(v.Value) <= SqlInt64.MaxValue / Math.Abs(si.Value)) _
          Then
            si = si * v
        Else
            si = 0
        End If
    End Sub

    Public Sub Merge(ByVal Group As VBProdAgg)
        Accumulate(Group.Terminate())
    End Sub

    Public Function Terminate() As SqlInt64
        If si.IsNull = True Then
            Return SqlInt64.Null
        Else
            Return si
        End If
    End Function

End Class
```

להלן ההנחיות צעד–אחר–צעד לפיהן תיצור את ה–assemblies ב– Visual Studio 2005.

יצירת assembly ב– Visual Studio 2005

1. ב– Visual Studio 2005, צור פרויקט C# חדש. השתמש בתיקייה Database ובתבנית SQL Server Project.

שים לב: תבנית זו אינה זמינה ב– Visual Studio 2005, Standard edition. אם אתה עובד בגרסה זו, השתמש בתבנית Class Library וכתוב ידנית את כל הקוד.

2. בתיבת הדו–שיח New Project, הגדר את המידע הבא:

Name: CSUDAs

Location: C:\

Solution Name: UDAs

כשתסיים להזין את המידע, אשר שהוא נכון.

3. בשלב זה, תתבקש לציין קישור למסד נתונים. צור קישור חדש למסד נתונים עבור מסד הנתונים tempdb במופע של SQL Server איתו אתה עובד, ובחר בו. הקישור למסד הנתונים בו אתה בוחר, מורה ל– Visual Studio היכן להטמיע את פונקציית ה–UDA שאתה מפתח.

4. לאחר אישור הבחירה של הקישור למסד הנתונים, תופיע תיבת שאלה שתשאל אותך האם ברצונך לאפשר איתור שגיאות SQL/CLR בקישור זה. בחר "לא". דוגמאות ה–UDA שתבנה בפרק זה פשוטות למדי, ולא יהיה צורך באיתור שגיאות.

5. בחלון Solution Explorer, לחץ בלחצן הימני בעכבר על הפרויקט CSUDAs, בחר בפריטי התפריט Add ו–aggregate, ואז בחר בתבנית Aggregate. שנה את שם המחלקה Aggregate1.cs ל– **CSUDAs_Classes.cs** ואשר.

6. בחן את הקוד של התבנית. תמצא שפונקציית UDA מיושמת כמבנה (struct ב– #C, structure ב– Visual Basic .NET). היא יכולה להיות מיושמת גם כמחלקה. הקטע הראשון של הקוד בתבנית מכיל namespaces הנמצאים בשימוש ב–assembly (שורות קוד המתחילות ב–"using"). הוסף עוד שלוש שורות קוד שיכילו את ה–namespaces הבאים: System.Text, System.IO ו– System.Runtime.InteropServices. (תוכל להעתיק אותם מקטע–קוד 8–6). אתה עומד להשתמש במחלקה StringBuilder מה– namespace: System.Text, במחלקות BinaryReader ו–BinaryWriter מה– namespace: System.IO, ולבסוף במאפיין StructLayout מה– namespace: System.Runtime.InteropServices (בפונקציית ה–UDA השנייה).

7. שנה את שם ברירת המחדל של פונקציית ה–UDA – שכרגע זהה לשם המחלקה (CSUDAs_Classes) – ל–**CSStrAgg**.

8. בתבנית תמצא ארבע שיטות המסופקות כבר איתה. אלו הן השיטות שכל פונקציית UDA חייבת ליישם. עם זאת, אם הנך משתמש בתבנית Class Library עבור הפרויקט שלך, תאלץ לכתוב אותם ידנית. תוך שימוש בתבנית Aggregate, כל שעליך לעשות הוא למלא אותם בקוד שלך. להלן תיאור של ארבע השיטות:

- Init: שיטה זו משמשת לאתחול החישוב. היא מופעלת פעם אחת לכל קבוצה לה מבצע ה– query processor פונקציית צבירה.
- Accumulate: שם השיטה רומז לך על מטרתה – צבירת ערכי הצבירה כמובן. שיטה זו מופעלת פעם אחת לכל ערך (כלומר, לכל אחת מהשורות) בקבוצה שמבוצעת בה פונקציית צבירה. היא משתמשת בפרמטר קלט, והפרמטר צריך

להיות מטיפוס נתונים התואם את טיפוס הנתונים של הטור שאתה עומד לבצע בו פונקציית צבירה ב– SQL Server. טיפוס הנתונים של הקלט יכול להיות גם CLR UDT.

- Merge: תוכל להבחין ששיטה זו משתמשת בפרמטר קלט מסוג מחלקת הצבירה. השיטה משמשת למיזוג מספר חישובים חלקיים של צבירה.
- Terminate: שיטה זו מסיימת את פונקציית הצבירה ומחזירה את התוצאה.

9. הוסף שני משתנים פנימיים (פרטיים) – sb ו–firstConcat – למחלקה לפני השיטה Init. אתה יכול לבצע זאת על ידי העתקה של הקוד המצהיר עליהם בקטע–קוד 8–6. המשתנה sb הוא מסוג StringBuilder ויחזיק את ערך הביניים של פונקציית הצבירה. המשתנה firstConcat הוא מסוג Boolean ומשמש להגדרה האם מחרוזת הקלט היא הראשונה אותה אתה משרשר בקבוצה. עבור כל ערכי הקלט מלבד הראשון, אתה עומד להוסיף פסיק לפני הערך שאתה משרשר.

10. החלף את הקוד הנוכחי של ארבע השיטות עם הקוד שמיישם אותן בקטע–קוד 8–6. זכור את הנקודות הבאות לכל שיטה:
    - בשיטה Init, אתה מאתחל את sb עם מחרוזת ריקה ואת firstConcat עם true.
    - בשיטה Accumulate, שים לב שאם ערך הפרמטר הוא NULL, הערך המצטבר יהיה NULL גם הוא. כמו כן, שים לב לטיפול השונה בערך הראשון, שזה עתה נוסף, ובערכים הבאים, עבורם ישנה תוספת של פסיק מקדים.
    - בשיטה Merge, אתה פשוט מוסיף פונקציית צבירה חלקית לזו הנוכחית. אתה עושה זאת על ידי קריאה לשיטה Accumulate של פונקציית הצבירה הנוכחית, והוספת הסיומת (ערך סופי) של פונקציית הצבירה החלקית האחרת. הקלט של הפונקציה Merge מתייחס לשם המחלקה ששינית קודם ל–CSStrAgg.
    - השיטה Terminate גם היא פשוטה מאוד; היא מחזירה את ייצוג המחרוזת של ערך הצבירה.

11. מחק את שתי השורות האחרונות במחלקה של הקוד מהתבנית; אלו שומרי מקום של שדה איבר. את כל שדות האיברים להם אתה זקוק הגדרת כבר בתחילת פונקציית ה–UDA.

12. כעת, חזור לתחילת פונקציית UDA, מייד לאחר הוספת ה–namespaces. תמצא שמות מאפיינים (attributes) שתרצה להוסיף. מאפיינים מסייעים ל– Visual Studio בהטמעה, ומסייעים ל– SQL Server ליעל את השימוש ב–UDA. פונקציות UDA חייבות לכלול את המאפיין Serializable. Serialization ב–NET. פירושו שמירה תמידית של ערכי השדות במחלקה. פונקציות UDA זקוקות ל– serialization עבור תוצאות ביניים. מבנה ה– serialization יכול להיות מקורי/מובנה (native), כלומר נקבע על ידי SQL Server, או מוגדר משתמש. Serialization יכול להיות מקורי אם אתה משתמש רק בטיפוסי ערכים של NET.; עליו להיות מוגדר משתמש אם הנך משתמש בטיפוסי מצביעים של NET.. לרוע המזל, הטיפוס string הוא טיפוס מצביע ב–NET.. לפיכך, עליך

להכין serialization משל עצמך. עליך ליישם את הממשק IBinarySerialize, המגדיר רק שתי שיטות: Read ו–Write. היישום של שיטות אלו ב–UDA שלנו פשוט ביותר. השיטה Read משתמשת בשיטה ReadString של המחלקה StringBuilder. השיטה Write משתמשת בשיטת ברירת המחדל ToString. את השיטה ToString יורשות כל מחלקות ה–NET. מהמחלקה העליונה ביותר, הנקראת SystemObject.

המשך ליישם את פונקציית ה–UDA לפי הצעדים הבאים:

א. הגדר שבכוונתך ליישם את הממשק IBinarySerialize במבנה. אתה עושה זאת על ידי הוספת נקודתיים ואת שם הממשק מייד לאחר שם המבנה (שם ה–UDA).

ב. העתק את השיטות Read ו–Write מקטע–קוד 8–6 לסוף ה–UDA שלך.

ג. שנה את התכונה Format.Native של המאפיין SqlUserDefinedAggregate ל– **Format.UserDefined**. עם serialization מוגדר–משתמש, פונקציית הצבירה שלך מוגבלת ל–8000 בתים בלבד. עליך לציין מה המספר המרבי של בתים שפונקציית ה–UDA שלך עשויה להחזיר עם התכונה MaxByteSize של המאפיין SqlUserDefinedAggregate. כדי לקבל את אורך המחרוזת האפשרי המרבי, הגדר MaxByteSize = 8000.

13. בקטע–קוד 8–6 תמצא עוד תכונות מעניינות של המאפיין SqlUserDefinedAggregate. הבה נבחן אותן:

- IsInvariantToDuplicates: זוהי תכונה אופציונלית. לדוגמה, פונקציית הצבירה MAX אדישה לכפילויות, בעוד שפונקציית הצבירה SUM אינה אדישה.
- IsInvariantToNulls: זוהי תכונה אופציונלית נוספת. היא מציינת אם פונקציית הצבירה אדישה ל– NULLs.
- IsInvariantToOrder: תכונה זו שמורה לשימוש עתידי. כרגע ה– query processor מתעלם ממנה ולכן מיון אינו מובטח.
- IsNullIfEmpty: תכונה זו מציינת אם פונקציית הצבירה תחזיר NULL במידה ולא הצטברו ערכים.

14. הוסף את התכונות המוזכרות לעיל ל–UDA שלך על ידי העתקתם מקטע–קוד 8–6. כעת השלמת את ה–UDA הראשון שלך!

15. בקטע–קוד 8–6 יש גם את הקוד ליישום UDA של מכפלה (CSProdAgg). העתק את הקוד המלא המיישם CSProdAgg לקוד שלך. שים לב שפונקציית ה–UDA הזו כוללת טיפול בטיפוס big integer בלבד. מכיוון שפונקציית ה–UDA היא פנימית ומטפלת רק בטיפוסי ערכים, היא יכולה להשתמש ב– serialization מקורי. serialization מקורי דורש שה– StructLayoutAttribute יוגדר כ– StructLayout.LayoutKindSequential אם פונקציית ה–UDA מוגדרת במחלקה ולא במבנה. אחרת, פונקציית ה–UDA מיישמת את אותן ארבע שיטות כמו פונקציית ה–UDA הקודמת שלך. קיימת בדיקה נוספת בשיטה Accumulate שמונעת ערכים מחוץ לתחום.

16. לבסוף, הוסף את גרסת ה– Visual Basic .NET של שתי פונקציות ה–UDA שנוצרו עד עתה:

א. מתפריט File, בחר בפריטים Add ו– New Project כדי לטעון את תיבת הדו–שיח Add New Project. נווט דרך סוג הפרויקט Visual Basic והתיקייה Database, ובחר ב– SQL Server Project. אל תאשר עדיין.

ב. בתיבת הדו–שיח Add New Project, הגדר את Name כ–**VBUDAs** ומיקום כ–**C:**. כעת אשר שהמידע נכון.

ג. השתמש באותו קישור מסד נתונים שיצרת עבור פרויקט ה–C# (הקישור ל–tempdb). השם של קישור מסד הנתונים שיצרת מוקדם יותר צריך להיות instancename.tempdb.dbo.

ד. בחלון Solution Explorer, הצבע עם המקש הימני של העכבר על הפרויקט VBUDAs, בחר Add ובחר בתבנית Aggregate. לפני שתאשר, שנה את שם המחלקה Aggregate1.vb ל– **VBUDAs_Classes.vb**.

ה. החלף את כל הקוד ב– VBUDAs_Classes.vb עם קוד ה– Visual Basic .NET המיישם את פונקציות ה–UDA מקטע–קוד 9–6.

17. שמור את כל הקבצים על ידי בחירה בפריט File שבתפריט ואז Save All.

18. צור את ה–assemblies על ידי בניית הפתרון. אתה עושה זאת על ידי בחירה בפריט Build בתפריט ואז Build Solution.

19. לבסוף, הטמע את הפתרון על ידי בחירה מהתפריט של הפריט Build ואז Deploy Solution.

בשלב זה שני ה–assemblies צריכים להיות מקוטלגים וכל ארבע פונקציות ה–UDA נוצרו כבר. כל הצעדים הללו הסתיימו אם אתה מטמיע את ה–assembly מ– Visual Studio .NET.

שים לב: כדי לעבוד עם פונקציות מבוססות–CLR ב– SQL Server, עליך לשנות את הגדרת השרת 'clr enabled' ל–1 (ברירת המחדל היא 0).

באפשרותך לבדוק האם ההטמעה הייתה מוצלחת על ידי דפדוף ב– sys.assemblies views ו– sys.assembly_modules, אשר במקרה שלנו נמצאים במסד הנתונים tempdb. כדי לאפשר CLR ולבצע שאילתות על views אלו, הרץ את הקוד בקטע–קוד 10–6.

קטע-קוד 10-6: אפשר CLR וביצוע שאילתות על views

```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE WITH OVERRIDE;
GO
USE tempdb;
GO
SELECT * FROM sys.assemblies;
SELECT * FROM sys.assembly_modules;
```

זה הכל. אתה משתמש בערכי פונקציית UDA כפי שאתה משתמש בכל פונקציית צבירה מובנית אחרת. כדי לבחון את הפונקציות החדשות, הרץ את הקוד הבא, ותקבל תוצאות זהות לאלו שהוחזרו על ידי הפתרונות האחרים לפונקציות צבירה מוגדרות-משתמש שהגדרתי קודם.

בדיקת פונקציות צבירה מוגדרות משתמש (UDA)

```
SELECT groupid, dbo.CSStrAgg(string) AS string
FROM tempdb.dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.VBStrAgg(string) AS string
FROM tempdb.dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.CSProdAgg(val) AS product
FROM tempdb.dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.VBProdAgg(val) AS product
FROM tempdb.dbo.Groups
GROUP BY groupid;
```

כשתסיים להתנסות עם UDA, הרץ את הקוד הבא כדי לנטרל את התמיכה ב-CLR:

```
EXEC sp_configure 'clr enabled', 0;
RECONFIGURE WITH OVERRIDE;
```

פתרונות ייחודיים

סוג אחר של פתרון לפונקציות צבירה מוגדרות-משתמש הוא פיתוח פתרון ייחודי אופטימלי לכל פונקציית צבירה. היתרון הוא לרוב הביצועים המשופרים של הפתרון. החיסרון הוא שסביר להניח שלא תוכל להשתמש בהיגיון דומה לחישובי פונקציות צבירה אחרים.

פתרון ייחודי לפונקציית הצבירה שרשור מחרוזות

פתרון ייחודי לפונקציית צבירה שרשור מחרוזות משתמש במצב PATH של אפשרות השאילתה FOR XML. שיטה יפה (ומהירה ביותר) זו פותחה על ידי מייקל ריס (Michael Rys), מנהל בצוות הפיתוח של Microsoft SQL Server שאחראי על טכנולוגיות SQL Server XML, ויוג'ין קוגן (Eugene Kogan), מנהל טכני בצוות של Microsoft SQL Server Engine. המצב PATH מספק דרך קלה יותר לשלב אלמנטים ומאפיינים מאשר ההנחיה EXPLICIT. להלן הפתרון הייחודי לפונקציית צבירה שרשור מחרוזות:

```
SELECT groupid,
  STUFF((SELECT ',' + string AS [text()]
         FROM dbo.Groups AS G2
         WHERE G2.groupid = G1.groupid
         ORDER BY memberid
         FOR XML PATH('')), 1, 1, '') AS string
FROM dbo.Groups AS G1
GROUP BY groupid;
```

תת–השאילתה למעשה מחזירה נתיב ממוין של כל המחרוזות בתוך הקבוצה הנוכחית. מכיוון שמחרוזת ריקה מסופקת לפסוקית PATH כקלט, אלמנט עוטף אינו מיוצר. הערכים המוחזרים מביטוי ללא שם טור תוצאה (למשל, ',' + string) או ביטוי עבורו מסופק הכינוי [text()], מוכנסים כאברי טקסט. המטרה של הפונקציה STUFF היא פשוט להסיר את הפסיק הראשון (על ידי החלפתו במחרוזת ריקה).

פתרון ייחודי לפונקציית הצבירה הכפלה

זכור שכדי לחשב את פונקציית הצבירה הכפלה (aggregate product) עליך לסרוק את כל הערכים בקבוצה. כך שפוטנציאל הביצועים של הפתרון שלך הוא להשיג את החישוב על ידי סריקה יחידה של הנתונים, תוך שימוש בשאילתה מבוססת–סטים. במקרה של פונקציית הצבירה הכפלה, ניתן להשיג זאת על ידי מניפולציה מתמטית המתבססת על לוגריתמים. אסתמך על משוואות הלוגריתמים הבאות:

משוואה 1: $\log_a(b) = x \text{ if and only if } a^x = b$

משוואה 2: $\log_a(v1 * v2 * \ldots * vn) = \log_a(v1) + \log_a(v2) + \ldots + \log_a(vn)$

למעשה, מה שאתה עומד לעשות כאן הוא טרנספורמציה של חישובים. ב–T-SQL קיימת תמיכה לפונקציות LOG, POWER ו–SUM. בעזרת שימוש באלו, באפשרותך לייצר את המכפלה החסרה. קבץ את הנתונים לפי הטור groupid, כפי שהיית עושה עם כל פונקציית צבירה מובנית. הביטוי SUM(LOG10(val)) מקביל לצד הימני של משוואה 2, כשהבסיס a שווה 10 במקרה שלנו, מכיוון שהשתמשת בפונקציה LOG10. להשגת מכפלת האלמנטים, כל שנותר לך לעשות הוא להעלות את הבסיס (10) בחזקת הצד הימני של

המשוואה. במילים אחרות, הביטוי POWER(10., SUM(LOG10(val))) נותן לך את מכפלת האלמנטים בקבוצה. כך נראית השאילתה המלאה:

```
SELECT groupid, POWER(10., SUM(LOG10(val))) AS product
FROM dbo.Groups
GROUP BY groupid;
```

זהו הפתרון הסופי אם אתה עוסק בערכים חיוביים בלבד. הפונקציה הלוגריתמית אינה מוגדרת למספרים שליליים ולאפס. באפשרותך להשתמש בשיטות סיבוב על ציר כדי לזהות ולהתמודד עם אפסים ושליליים כלהלן:

```
SELECT groupid,
  CASE
    WHEN MAX(CASE WHEN val = 0 THEN 1 END) = 1 THEN 0
    ELSE
      CASE WHEN COUNT(CASE WHEN val < 0 THEN 1 END) % 2 = 0
        THEN 1 ELSE -1
      END * POWER(10., SUM(LOG10(NULLIF(ABS(val), 0))))
  END AS product
FROM dbo.Groups
GROUP BY groupid;
```

ביטוי ה–CASE החיצוני משתמש ראשית בשיטת סיבוב על ציר כדי לבדוק האם מופיע בקבוצה הערך 0, שעבורו יחזיר 0 כתוצאה. הפסוקית ELSE קוראת לביטוי CASE נוסף, המשתמש גם הוא בשיטת סיבוב על ציר כדי לספור את מספר הערכים השליליים בקבוצה. אם המספר הוא זוגי, הוא מפיק 1+; אם הוא אי–זוגי, הוא מפיק 1-. המטרה של חישוב זה היא לקבוע את הסימן המתמטי של התוצאה. אז מוכפל הסימן (1- או 1+) במכפלת הערכים האבסולוטיים בקבוצה לקבלת המכפלה הרצויה.

שים לב שיש כאן שימוש ב–NULLIF להחלפת אפסים ב–NULLs. ייתכן שתצפה שחלק זה של הביטוי לא יוערך כלל אם נמצא אפס. אך זכור שה–optimizer יכול לשקול תוכניות פיסיות שונות רבות לביצוע השאילתה שלך. כתוצאה מכך, אינך יכול להיות בטוח מה הסדר האמיתי לפיו יוערכו חלקי הביטוי. על ידי החלפת אפסים ב–NULLs, אתה מבטיח שלעולם לא תקבל שגיאת domain אם הפונקציה LOG10 תופעל לבסוף עם אפס כקלט. שימוש זה ב–NULLIF, בשילוב עם השימוש ב–ABS, מאפשר לפתרון זה לספק פתרונות לכל סימן (שלילי, אפס וחיובי).

תוכל לפעול גם בגישה מתמטית טהורה לטיפול באפסים ובערכים שליליים על ידי שימוש בשאילתה הבאה:

```
SELECT groupid,
  CAST(ROUND(EXP(SUM(LOG(ABS(NULLIF(val,0)))))*
    (1-SUM(1-SIGN(val))%4)*(1-SUM(1-SQUARE(SIGN(val)))),0) AS INT)
```

```
 AS product
FROM dbo.Groups
GROUP BY groupid;
```

דוגמה זו מראה לך שלעולם אין לאבד תקווה כאשר מחפשים אחר פתרון יעיל. אם תשקיע זמן ותחשוב מחוץ לקופסה, ברוב המקרים, תמצא פתרון.

פתרון ייחודי לפונקציית הצבירה פעולות מבוססות–סיבית

כעת, אציג פתרונות ייחודיים לפונקציית צבירה של פעולות מבוססות–סיבית (bitwise) ב–T-SQL – פעולת הסיבית OR (|), פעולת הסיבית AND (&) ופעולת הסיבית XOR (^). אניח שאתה מכיר את העקרונות של פעולות מבוססות–סיבית והשימוש בהן, ואספק סריקה קצרה בלבד. אם אינך מכיר אותן, אנא פנה ראשית לפרק "Bitwise Operators" ב– Books Online.

פעולות מבוססות–סיבית הן פעולות המבוצעות על הסיביות הבודדות של נתוני מספרים שלמים. לכל סיבית קיימים שני ערכים אפשריים, 1 ו–0. מספרים שלמים יכולים לשמש לאחסנת מפות סיביות או מחרוזות של סיביות, ולמעשה הם משמשים בתוכנה SQL Server לאחסנת מידע מסוג metadata – למשל, תכונות של אינדקסים (clustered, ייחודי וכד') ותכונות של מסדי נתונים (קריאה בלבד, גישה מוגבלת, כיווץ אוטומטי, וכד'). ייתכן שתבחר גם לאחסן מפות סיביות בעצמך לייצוג סטים של מאפיינים בינאריים – למשל, סט הרשאות כאשר כל סיבית מייצגת הרשאה שונה.

ישנם מומחים שממליצים לא להשתמש בעיצוב כזה מכיוון שהוא מפר את 1NF (חוק נורמליזציה ראשון – אין קבוצות שחוזרות על עצמן). ייתכן שתעדיף לעצב את הנתונים שלך בצורה יותר מנורמלת, כאשר מאפיינים כמו זה מאוחסנים בטורים נפרדים. איני מעוניין להיכנס לדיון איזה עיצוב עדיף. כאן אניח עיצוב נתון שכן מאחסן מפות סיביות עם סטים של דגלים, ואניח שעליך לבצע פעולות צבירה (אגרגציה) מבוססות–סיביות על מפות סיביות אלו. אני מעוניין רק להציג את השיטות למקרים בהם יש לך צורך להשתמש בהם.

פעולת הסיבית OR (|) משמשת לרוב לבניית מפות סיביות או לייצור מפת סיביות תוצאתית הצוברת את כל הסיביות הדלוקות. בתוצאה של פעולת הסיבית OR, סיביות דלוקות (כלומר, בעלות ערך 1) אם הן דלוקות לפחות באחת ממפות הסיביות הנפרדות.

פעולת הסיבית AND (&) משמשת לרוב לבדיקה האם סיבית מסוימת (או סט סיביות) דלוקה על ידי פעולת AND בין מפת הסיביות המקורית למסכת סיביות. היא משמשת גם לצבירה של סיביות דלוקות בלבד בכל מפות הסיביות. היא מייצרת סיבית תוצאה הדלוקה כאשר סיבית זו דלוקה בכל מפות הסיביות הנפרדות.

פעולת הסיבית XOR (^) משמשת לרוב לחישוב parity או כחלק מתוכנית להצפנת נתונים. לכל מיקום סיבית, סיבית התוצאה נדלקת אם היא דלוקה במספר אי–זוגי של מפות הסיביות הנפרדות.

שים לב: פעולת הסיבית XOR היא האופרטור מבוסס–הסיביות ההפיך היחיד. זו הסיבה שהיא משמשת לחישובי parity ולהצפנה.

גרסאות צבירה (אגרגציה) של האופרטורים מבוססי–הסיביות אינן קיימות ב– SQL Server, וכאן אספק פתרונות לביצוע פונקציות צבירה מבוססות–סיבית. אשתמש באותה טבלת Groups בה השתמשתי בדוגמאות האחרות לפונקציות צבירה מוגדרות–משתמש שנתתי. הנח שטור המספרים השלמים val מייצג מפת סיביות. כדי לראות את ייצוג הסיביות של כל שלם, ראשית צור את הפונקציה fn_dectobase על ידי הרצת הקוד בקטע–קוד 6–11.

קטע–קוד 6–11: קוד ליצירת הפונקציה fn_dectobase

```
IF OBJECT_ID('dbo.fn_dectobase') IS NOT NULL
  DROP FUNCTION dbo.fn_dectobase;
GO
CREATE FUNCTION dbo.fn_dectobase(@val AS BIGINT, @base AS INT)
  RETURNS VARCHAR(63)
AS
BEGIN
  IF @val < 0 OR @base < 2 OR @base > 36 RETURN NULL;
  DECLARE @r AS VARCHAR(63), @alldigits AS VARCHAR(36);

  SET @alldigits = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';

  SET @r = '';
  WHILE @val > 0
  BEGIN
    SET @r = SUBSTRING(@alldigits, @val % @base + 1, 1) + @r;
    SET @val = @val / @base;
  END

  RETURN @r;
END
GO
```

הפונקציה מקבלת שני קלטים: מספר שלם בעל 64–סיביות המחזיק את מפת הסיביות המקורית, ובסיס בו אתה מעוניין לייצג את הנתונים. השתמש בשאילתה הבאה כדי להחזיר

את ייצוג הסיביות של המספרים השלמים בטור val בטבלה Groups. התוצאה, בצורה מקוצרת (10 הספרות הימניות בלבד של binval), מוצגת בטבלה 22–6:

```
SELECT groupid, val,
  RIGHT(REPLICATE('0', 32) + CAST(dbo.fn_dectobase(val, 2) AS
        VARCHAR(64)), 32) AS binval
FROM dbo.Groups;
```

טבלה 22–6: ייצוג בינארי של ערכים

groupid	*val*	*binval*
a	6	0000000110
a	7	0000000111
b	3	0000000011
b	7	0000000111
b	3	0000000011
b	11	0000001011
c	8	0000001000
c	10	0000001010
c	12	0000001100

הטור binval מציג את הטור val בבסיס 2, עם אפסים מובילים ליצירת מחרוזת בעלת מספר קבוע של ספרות. כמובן שבאפשרותך לשנות את מספר האפסים המובילים בהתאם לצרכיך. בדוגמאות הקוד שלי, לא שילבתי את הקריאה לפונקציה זו כדי שדעתך לא תוסח מהטכניקות בהן רציתי להתמקד. אך כן קראתי לה לייצור ייצוג הסיביות בכל הפלטים שאציג.

פונקציית הצבירה של פעולת סיבית OR

ללא עיכובים נוספים, הבה נתחיל בחישוב פונקציית הצבירה של פעולת סיבית OR. כדי לתת הקשר מוחשי לבעיה, תאר לך שאתה מתחזק אבטחת יישומים במסד הנתונים. הטור groupid מייצג משתמש, והטור val מייצג מפת סיביות עם מצבי הרשאה (1 עבור מורשה או 0 עבור לא–מורשה) של תפקיד (role) אליו שייך המשתמש. אתה מעוניין במפת הסיביות בפועל לכל משתמש (קבוצה), שצריכה להיות מחושבת כפונקציית צבירה של פעולת הסיביות OR בין כל מפות הסיביות של התפקידים אליהם שייך המשתמש.

ההיבט העיקרי של פעולת סיביות OR עליו אשען בפתרונות שלי הוא העובדה שהוא שווה לסכום האריתמטי של הערכים המיוצגים על ידי כל ערך סיבית ייחודי הדלוק

במפות הסיביות השונות. בתוך מספר שלם, סיבית מייצגת את הערך 2^(bit_pos-1). למשל, ערך הסיבית של הסיבית השלישית הוא $2^2 = 4$. קח למשל את מפות הסיביות עבור משתמש c: (1000) 8, (1010) 10 ו–(1100) 12. במפת הסיביות 8 יש סיבית אחת בלבד דלוקה – ערך הסיבית המייצג 8, ב–10 דלוקות הסיביות המייצגות 2 ו–8, וב–12 דלוקות הסיביות המייצגות 4 ו–8. הסיביות הייחודיות הדלוקות בכל אחד מהמספרים השלמים 8, 10 ו–12 הן סיביות 2, 4 ו–8, כך שפונקציית הצבירה של פעולת הסיבית OR של 8, 10 ו–12 שווה ל: (1110) 14 = 8 + 4 + 2.

הפתרון הבא נשען על ההיגיון לעיל על ידי חילוץ ערכי הסיביות האינדיבידואליות הדלוקים בכל אחת ממפות הסיביות המשתתפות. החילוץ מושג על ידי שימוש בביטוי MAX(val & <bitval>). השאילתה אז מבצעת סכום אריתמטי של ערכי הסיביות האינדיבידואליות:

```
SELECT groupid,
    MAX(val & 1)
  + MAX(val & 2)
  + MAX(val & 4)
  + MAX(val & 8)
-- ...
  + MAX(val & 1073741824) AS agg_or
FROM dbo.Groups
GROUP BY groupid;
```

התוצאה של צבירת פעולת הסיביות OR מוצגת בטבלה 23–6, כולל 10 הספרות הימניות של הייצוג הבינארי של ערך התוצאה.

טבלה 23–6: אגרגציית פעולת סיביות OR

groupid	*agg_or*	*agg_or_binval*
a	7	0000000111
b	15	0000001111
c	14	0000001110

בדומה, תוכל להשתמש ב– SUM(DISTINCT val & <bitval>) במקום ב– MAX(val & <bitval>), מכיוון שהתוצאות האפשריות היחידות הן <bitval> ו–0:

```
SELECT groupid,
     SUM(DISTINCT val & 1)
  + SUM(DISTINCT val & 2)
  + SUM(DISTINCT val & 4)
  + SUM(DISTINCT val & 8)
-- ...
  + SUM(DISTINCT val & 1073741824) AS agg_or
FROM dbo.Groups
GROUP BY groupid;
```

שני הפתרונות סובלים מאותה מגבלה – מחרוזות שאילתה ארוכות – בגלל הצורך בביטוי שונה לכל ערך סיבית. במאמץ לקצר את מחרוזות השאילתה, באפשרותך להשתמש בטבלת עזר. אתה מאחד את הטבלה Groups עם טבלת עזר המכילה את כל ערכי הסיביות הרלוונטיות, תוך שימוש ב– val & bitval = bitval כתנאי ה–join. תוצאת האיחוד תכלול את כל ערכי הסיביות הדלוקות בכל אחת ממפות הסיביות. אז תוכל למצוא עבור כל קבוצה (SUM(DISTINCT <bitval>. את טבלת העזר לערכי סיבית ניתן לייצר בקלות מטבלת Nums בה השתמשנו קודם. סנן מספרים ככמות הסיביות לה תזדקק, והעלה 2 בחזקת n-1. להלן הפתרון המלא:

```
SELECT groupid, SUM(DISTINCT bitval) AS agg_or
FROM dbo.Groups
  JOIN (SELECT POWER(2, n-1) AS bitval
        FROM dbo.Nums
        WHERE n <= 31) AS Bits
    ON val & bitval = bitval
GROUP BY groupid;
```

פונקציית הצבירה של פעולת סיבית AND

בצורה דומה באפשרותך לחשב צבירה (אגרגציה) של פעולת סיבית AND. בתרחיש ההרשאות, פונקציית הצבירה פעולת סיבית AND תייצג את סט ההרשאות המוגבל ביותר. זכור רק שערך סיבית צריך להתווסף לסכום האריתמטי אך ורק אם היא דלוקה בכל מפות הסיביות. כך שעליך לקבץ תחילה את הנתונים לפי groupid ו–bitval, וסנן רק את הקבוצות בהן MIN(val & bitval) > 0, כלומר שהסיבית הייתה דלוקה בכל מפות הסיביות שלהן. בשאילתה חיצונית, קבץ את הנתונים לפי groupid ובצע את הסכום האריתמטי של ערכי הסיבית מהשאילתה הפנימית:

```
SELECT groupid, SUM(bitval) AS agg_and
FROM (SELECT groupid, bitval
      FROM dbo.Groups,
        (SELECT POWER(2, n-1) AS bitval
```

```
         FROM dbo.Nums
         WHERE n <= 31) AS Bits
      GROUP BY groupid, bitval
      HAVING MIN(val & bitval) > 0) AS D
GROUP BY groupid;
```

התוצאה של צבירת פעולת הסיבית AND מוצגת בטבלה 6–24.

טבלה 6–24: פונקציית הצבירה של פעולת סיבית AND

groupid	*agg_and*	*agg_and_binval*
a	6	0000000110
b	3	0000000011
c	8	0000001000

פונקציית הצבירה של פעולת הסיבית XOR

כדי לחשב צבירה של פעולת סיבית XOR, סנן רק את קבוצות groupid, bitval להן מספר אי–זוגי של סיביות דלוקות כפי שמוצג בקוד הבא, המדגים פונקציית צבירה של פעולת סיבית XOR תוך שימוש ב–Nums ומייצר את הפלט המוצג בטבלה 6–25:

```
SELECT groupid, SUM(bitval) AS agg_xor
FROM (SELECT groupid, bitval
      FROM dbo.Groups,
        (SELECT POWER(2, n-1) AS bitval
         FROM dbo.Nums
         WHERE n <= 31) AS Bits
      GROUP BY groupid, bitval
      HAVING SUM(SIGN(val & bitval)) % 2 = 1) AS D
GROUP BY groupid;
```

טבלה 6–25: פונקציית הצבירה של פעולת הסיבית XOR

groupid	*agg_xor*	*agg_xor_binval*
a	1	0000000001
b	12	0000001100
c	14	0000001110

חציון

כדוגמה אחרונה לפתרון ייחודי לפונקציית צבירה מוגדרת משתמש, אשתמש בחישוב הסטטיסטי של חציון (median). נניח שעליך לחשב את החציון של טור val לכל קבוצה. לחציון קיימות שתי הגדרות שונות. כאן נחזיר את הערך האמצעי במקרה של מספר אי–זוגי של אלמנטים, ואת הממוצע של שני הערכים האמצעיים במקרה של מספר זוגי של אלמנטים.

הקוד הבא מציג שיטה לחישוב החציון, ומפיק את הפלט המוצג בטבלה 26–6:

```
WITH Tiles AS
(
  SELECT groupid, val,
    NTILE(2) OVER(PARTITION BY groupid ORDER BY val) AS tile
  FROM dbo.Groups
),
GroupedTiles AS
(
  SELECT groupid, tile, COUNT(*) AS cnt,
    CASE WHEN tile = 1 THEN MAX(val) ELSE MIN(val) END AS val
  FROM Tiles
  GROUP BY groupid, tile
)
SELECT groupid,
  CASE WHEN MIN(cnt) = MAX(cnt) THEN AVG(1.*val)
       ELSE MIN(val) END AS median
FROM GroupedTiles
GROUP BY groupid;
```

טבלה 26–6: חציון

groupid	*median*
a	6.500000
b	5.000000
c	10.000000

ה– Tiles CTE מחשב את הערך NTILE(2) בתוך קבוצה, בהתבסס על מיון val. כאשר ישנו מספר זוגי של אלמנטים, המחצית הראשונה של הערכים תקבל מספר אריח 1 והמחצית השנייה תקבל מספר אריח 2. במקרה זוגי, החציון אמור להיות הממוצע של הערך הגבוה ביותר בתוך האריח הראשון והנמוך ביותר בשני. כאשר ישנו מספר אי–זוגי

של אלמנטים, זכור ששורה נוספת מתווספת לקבוצה הראשונה. המשמעות היא שהערך הגבוה ביותר באריח הראשון הוא החציון.

ה–CTE השני (GroupedTiles) מקבץ את הנתונים לפי מספר קבוצה ואריח, ומחזיר את ספירת השורות לכל קבוצה ואריח וכן את הטור val, אשר עבור האריח הראשון הוא הערך המקסימלי בתוך האריח ועבור האריח השני הוא הערך המינימלי בתוך האריח.

השאילתה החיצונית מקבצת את שתי השורות בכל קבוצה (אחת מייצגת כל אריח). ביטוי CASE ברשימת ה–SELECT קובע מה להחזיר בהתבסס על ה–parity של ספירת השורות בקבוצה. כאשר לקבוצה יש מספר זוגי של שורות (כלומר, לשני האריחים של הקבוצה יש אותה ספירת שורות), אתה מקבל את הממוצע של המקסימום באריח הראשון והמינימום בשני. כאשר לקבוצה יש מספר אי–זוגי של אלמנטים (כלומר, לשני האריחים של הקבוצה יש מספר שורות שונה), אתה מקבל את המינימום מבין שני הערכים, שהוא גם המקסימום באריח הראשון, שבתורו הוא גם החציון.

בעזרת שימוש בפונקציה ROW_NUMBER, ניתן למצוא פתרונות נוספים למציאת החציון, אלגנטיים יותר ומעט פשוטים יותר. להלן הדוגמה הראשונה:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER()
      OVER(PARTITION BY groupid ORDER BY val, memberid) AS rna,
    ROW_NUMBER()
      OVER(PARTITION BY groupid ORDER BY val DESC, memberid DESC) AS rnd
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE ABS(rna - rnd) <= 1
GROUP BY groupid;
```

הרעיון הוא לחשב שני מספרי שורה עבור כל שורה: אחד בהתבסס על val ואז memberid (שובר–השוויון) בסדר עולה (rna) והשני בהתבסס על אותם מאפיינים בסדר יורד (rnd). קיים יחס מתמטי מעניין בין שני רצפים ממוינים בכיוונים הפוכים אותו תוכל לנצל למטרתך. ההפרש המוחלט בין השניים הוא קטן–שווה 1 רק עבור האלמנטים שצריכים להשתתף בחישוב החציון. קח, למשל, קבוצה בעלת מספר אי–זוגי של אלמנטים; ABS(rna - rnd) שווה ל–0 רק עבור השורה האמצעית. עבור כל השורות האחרות הוא גדול מ–1. בהינתן מספר זוגי של אלמנטים, ההפרש הוא 1 עבור שתי השורות האמצעיות וגדול מ–1 עבור כל האחרות.

הסיבה לשימוש ב–memberid כשובר–שוויון היא כדי להבטיח דטרמיניזם של חישובי מספרי השורה. מכיוון שאתה מחשב שני מספרי שורה שונים, תרצה להבטיח שהערך המופיע במיקום n מההתחלה בסדר עולה יופיע במיקום n מהסוף בסדר יורד.

ברגע שהערכים שצריכים להשתתף בחישובי החציון מבודדים, כל שעליך לעשות הוא לקבץ אותם לפי groupid ולחשב את הממוצע בכל קבוצה.

באפשרותך להימנע מהצורך לחשב שני מספרי שורה נפרדים על ידי הפקת השני מתוך הראשון. מספרי השורה היורדים ניתנים לחישוב על ידי הפחתת מספרי השורה העולים מספירת השורות בקבוצה והוספת אחד. למשל, בקבוצה בעלת ארבעה אלמנטים, השורה שקיבלה מספר שורה עולה 1 תקבל את מספר השורה היורד 4 = 1 + 1 – 4. מספר שורה עולה 2 תקבל את מספר השורה היורד 3 = 1 + 2 – 4, וכך הלאה. הפקת מספר השורה היורד מזה העולה מבטל את הצורך בשובר–שוויון. אינך מתמודד עם שני חישובים נפרדים; לפיכך מתבטלת סוגית חוסר הדטרמיניזם.

כך שהחישוב rna - rnd הופך לחישוב הבא: rn - (cnt-rn+1) = 2*rn - cnt - 1. להלן שאילתה המיישמת היגיון זה:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY val) AS rn,
    COUNT(*) OVER(PARTITION BY groupid) AS cnt
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE ABS(2*rn - cnt - 1) <= 1
GROUP BY groupid;
```

קיימת דרך אחרת למציאת השורות המשתתפות בחישוב החציון בהתבסס על מספר שורה וספירת השורות בקבוצה: rn IN((cnt+1)/2, (cnt+2)/2). עבור מספר אי–זוגי של אלמנטים, שני הביטויים מפיקים את מספר השורה האמצעית. למשל, אם קיימות 7 שורות, הן הביטוי (7+1)/2 והן הביטוי (7+2)/2 שווים 4. עבור מספר זוגי של אלמנטים, הביטוי הראשון מפיק את מספר השורה מייד לפני נקודת האמצע והשני מפיק את מספר השורה מייד אחריה. אם קיימות 8 שורות, (8+1)/2 מפיק 4 ו–(8+2)/2 מפיק 5. להלן השאילתה המיישמת היגיון זה:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY val) AS rn,
    COUNT(*) OVER(PARTITION BY groupid) AS cnt
```

```
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE rn IN((cnt+1)/2, (cnt+2)/2)
GROUP BY groupid;
```

היסטוגרמות

היסטוגרמות הן כלים ניתוחיים חזקים ביותר המבטאים פיזור של פריטים. למשל, נניח שעליך למצוא מתוך נתוני ההזמנה בטבלת Orders כמה הזמנות קטנות, בינוניות וגדולות קיימות, בהתבסס על כמות ההזמנה. במילים אחרות, נדרשת לך היסטוגרמה עם שלוש מדרגות. מה שמגדיר הזמנות כקטנות, בינוניות או גדולות הן הכמויות הקיצוניות (כמויות המינימום והמקסימום). בטבלת Orders שלנו, כמות ההזמנה המינימלית היא 10 והמקסימלית היא 40. קח את ההפרש בין שני הקיצונים (30 = 10 - 40), וחלק אותו במספר המדרגות (3) לקבלת גודל המדרגה. במקרה שלנו, ההפרש הוא 30 חלקי 3 שווה 10. כך שהגבולות של מדרגה 1 (קטנה) יהיו 10 ו–20; של מדרגה 2 (בינונית) הם יהיו 20 ו–30; ולמדרגה 3 (גדולה) הם יהיו 30 ו–40.

לצורך הכללה, הנח mn = MIN(qty) ו– mx = MAX(qty). כמו כן הנח stepsize = (mx - mn) / @numsteps. בהינתן מספר מדרגה n, הגבול התחתון של המדרגה (lb) הוא mn + (n - 1) * stepsize והגבול העליון (hb) הוא mn + n * stepsize. נשאלת השאלה, באיזה ביטוי תשתמש לתחום את האלמנטים השייכים לכל מדרגה? אינך יכול להשתמש ב– qty BETWEEN lb and hb מכיוון שערך השווה ל–hb יופיע במדרגה זו וכן במדרגה הבאה, בה הוא יהיה שווה לגבול התחתון. זכור שאותו חישוב הפיק את הגבול העליון של מדרגה אחת ואת הגבול התחתון של המדרגה הבאה. גישה אחת להתמודדות עם בעיה זו היא להגדיל כל אחד מהגבולות התחתונים באחד, כך שיהיה גדול יותר מהגבול העליון של המדרגה הקודמת. הדבר אפשרי כאשר מדובר במספרים שלמים, אך עם טיפוס נתונים אחר זה לא יעבוד מכיוון שעלולים להיות ערכים בין שתי המדרגות, אך לא בתוך אף אחת מהן – אפשר לומר, ערכים שנופלים בין הכיסאות.

מה שהייתי רוצה לעשות כדי לפתור את הבעיה הוא לשמור את אותו ערך בשני הגבולות, ובמקום להשתמש ב–BETWEEN, אשתמש ב– qty >= lb ו– qty < hb. לשיטה זו יש גם סוגיות משל עצמה, אך אני מוצא שקל יותר להתמודד איתה מאשר עם השיטה הקודמת. הסוגיה כאן היא שהפריט עם הכמות הגדולה ביותר (40, במקרה שלנו) נשאר מחוץ להיסטוגרמה. כדי לפתור זאת, אני מוסיף שבר קטן מאוד לערך המקסימלי טרם ביצוע החישוב לגודל המדרגה: stepsize = ((1E0*mx + 0.0000000001) - mn) / @numsteps. זוהי שיטה המאפשרת לפריט בעל הערך הגבוה ביותר להיכלל, בעוד ההשפעה על ההיסטוגרמה תהיה זניחה. הכפלתי

את mx בערך הצף 1E0 להגנה מפני אובדן נקודת הנתונים העליונה כאשר qty הוא מטיפוס MONEY או SMALLMONEY.

ובכן, המרכיבים להם אתה נדרש ליצירת הגבולות העליון והתחתון של מדרגות ההיסטוגרמה הם: @numsteps (ניתן כקלט), מספר מדרגה (טור n מטבלת העזר Nums), mn ו–stepsize, אותם תיארתי קודם.

להלן קוד ה–T-SQL הנדרש לייצור מספר המדרגה, הגבול התחתון והגבול העליון של כל מדרגה בהיסטוגרמה, המפיק את הפלט המוצג בטבלה 27–6:

```
DECLARE @numsteps AS INT;
SET @numsteps = 3;

SELECT n AS step,
  mn + (n - 1) * stepsize AS lb,
  mn + n * stepsize AS hb
FROM dbo.Nums,
  (SELECT MIN(qty) AS mn,
     ((1E0*MAX(qty) + 0.0000000001) - MIN(qty))
     / @numsteps AS stepsize
   FROM dbo.Orders) AS D
WHERE n <= @numsteps;
```

טבלה 27–6: טבלת מדרגות היסטוגרמה

step	*lb*	*hb*
1	10	20.0000000000333
2	20.0000000000333	30.0000000000667
3	30.0000000000667	40.0000000001

ייתכן שתרצה לעטוף את הקוד הזה בפונקציה מוגדרת–משתמש כדי לפשט את השאילתות המחזירות את ההיסטוגרמה עצמה. הרץ את הקוד בקטע–קוד 12–6 כדי לבצע זאת.

קטע-קוד 6-12: קוד ליצירת הפונקציה fn_histsteps

```
CREATE FUNCTION dbo.fn_histsteps(@numsteps AS INT) RETURNS TABLE
AS
RETURN
  SELECT n AS step,
    mn + (n - 1) * stepsize AS lb,
    mn + n * stepsize AS hb
  FROM dbo.Nums,
    (SELECT MIN(qty) AS mn,
       ((1E0*MAX(qty) + 0.0000000001) - MIN(qty))
       / @numsteps AS stepsize
    FROM dbo.Orders) AS D
  WHERE n <= @numsteps;
GO
```

כדי לבחון את הפונקציה, הרץ את השאילתה הבאה, שתיתן לך טבלת מדרגות היסטוגרמה בת שלוש שורות:

```
SELECT * FROM dbo.fn_histsteps(3) AS S;
```

כדי להחזיר את ההיסטוגרמה עצמה, פשוט אחד את טבלת המדרגות ואת טבלת Orders עם הביטוי שתיארתי קודם לכן (qty >= lb AND qty < hb), קבץ את הנתונים לפי מספר מדרגה, והחזר את מספר המדרגה ומספר השורות:

```
SELECT step, COUNT(*) AS numorders
FROM dbo.fn_histsteps(3) AS S
  JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

שאילתה זו מייצרת את ההיסטוגרמה המוצגת בטבלה 6-28.

טבלה 6-28: היסטוגרמה בת שלוש מדרגות

step	*numorders*
1	8
2	2
3	1

תוכל לראות שקיימות שמונה הזמנות קטנות, שתי הזמנות בינוניות והזמנה גדולה אחת. כדי להחזיר היסטוגרמה בת 10 מדרגות, פשוט ספק 10 כקלט לפונקציה fn_histsteps, והשאילתה תפיק את ההיסטוגרמה המוצגת בטבלה 6–29:

```
SELECT step, COUNT(*) AS numorders
FROM dbo.fn_histsteps(10) AS S
  JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

טבלה 6–29: היסטוגרמה בת עשר מדרגות

step	*numorders*
1	4
2	2
4	3
7	1
10	1

שים לב שמכיוון שאתה משתמש ב–inner join, מדרגות ריקות אינן מוחזרות. כדי להחזיר גם מדרגות ריקות, תוכל להשתמש בשאילתה הבאה עם outer join, המפיקה את הפלט המוצג בטבלה 6–30:

```
SELECT step, COUNT(qty) AS numorders
FROM dbo.fn_histsteps(10) AS S
  LEFT OUTER JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

טבלה 6–30: היסטוגרמה בת עשר מדרגות, כולל מדרגות ריקות

step	*numorders*
1	4
2	2
3	0
4	3
5	0
6	0

step	*numorders*
7	1
8	0
9	0
10	1

שים לב: יש כאן שימוש ב– COUNT(qty) ולא ב– COUNT(*). COUNT(*) יחזיר באופן שגוי 1 עבור מדרגות ריקות מכיוון שבקבוצה קיימת שורה חיצונית. עליך לספק לפונקציה מאפיין מהצד הלא–שמור (Orders) כדי לקבל את הספירה הנכונה.

במקום להשתמש בשאילתה של outer join, תוכל להשתמש ב– cross join, עם מסנן המתאים הזמנות למדרגות, ובאפשרות GROUP BY ALL המבטיחה שגם מדרגות ריקות יוחזרו:

```
SELECT step, COUNT(qty) AS numcusts
FROM dbo.fn_histsteps(10) AS S, dbo.Orders AS O
WHERE qty >= lb AND qty < hb
GROUP BY ALL step;
```

רציתי רק להראות שניתן לכתוב פתרון פשוט יותר על ידי שימוש באפשרות GROUP BY ALL. אך זכור שעדיף להימנע מלהשתמש במאפיין זה, שאינו עומד בסטנדרט ונשאר במוצר מסיבות היסטוריות, שכן סביר להניח שהוא יוסר מהמוצר באחת מהגרסאות העתידיות.

קיימת אפשרות אחרת לטיפול בנושא גבולות המדרגה והביטוי המשמש לזיהוי התאמה. ניתן פשוט לבדוק אם מספר הצעד הוא 1, במקרה כזה אתה מפחית 1 מהגבול התחתון. אז, בשאילתה המפיקה את ההיסטוגרמה עצמה, אתה משתמש בביטוי qty > lb AND qty <= hb.

גישה אחרת היא לבדוק האם המדרגה היא האחרונה, ובמקרה שכן, להוסיף 1 לגבול העליון. אז השתמש בביטוי qty >= lb AND qty < hb.

קטע–קוד 13–6 מציג את הפונקציה המתוקנת המיישמת את הגישה האחרונה:

קטע–קוד 13–6: שינוי היישום של הפונקציה fn_histsteps

```
ALTER FUNCTION dbo.fn_histsteps(@numsteps AS INT) RETURNS TABLE
AS
RETURN
  SELECT n AS step,
    mn + (n - 1) * stepsize AS lb,
    mn + n * stepsize + CASE WHEN n = @numsteps THEN 1 ELSE 0 END AS hb
  FROM dbo.Nums,
```

```
   (SELECT MIN(qty) AS mn,
      (1E0*MAX(qty) - MIN(qty)) / @numsteps AS stepsize
    FROM dbo.Orders) AS D
  WHERE n < = @numsteps;
GO
```

והשאילתה הבאה מייצרת את ההיסטוגרמה עצמה:

```
SELECT step, COUNT(qty) AS numorders
FROM dbo.fn_histsteps(10) AS S
  LEFT OUTER JOIN dbo.Orders AS O
    ON qty >= lb AND qty < hb
GROUP BY step;
```

גורם הקבצה

בפרקים קודמים, במיוחד בפרק 4, תיארתי מושג לו קראתי **גורם הקבצה**. בפרט, השתמשתי בו בבעיה לבידוד איים, או תחומים של אלמנטים עוקבים בתוך רצף. זכור שגורם ההקבצה הוא הגורם שבו אתה משתמש לבסוף בפסוקית ה– GROUP BY שלך לזיהוי הקבוצה. בבעיה הקודמת, הצגתי שתי שיטות לחישוב גורם ההקבצה. שיטה אחת הייתה חישוב הערך המקסימלי בתוך קבוצה (בפרט, הערך הקטן ביותר שהוא גם גדול–שווה לערך הנוכחי וגם שלאחריו קיים פער). השיטה השנייה השתמשה במספרי שורה.

מכיוון שפרק זה עוסק בפונקציות צבירה (אגרגציות), הרי שמתאים כאן לבחון שוב את הבעיה המעשית הזו. בדוגמאות הבאות אשתמש בטבלה Stocks, שתיצור ותמלא בנתונים על ידי הרצת הקוד בקטע–קוד 14–6.

קטע–קוד 14–6: יצירה ומילוי של טבלת Stocks

```
USE tempdb;
GO
IF OBJECT_ID('Stocks') IS NOT NULL
  DROP TABLE Stocks;
GO

CREATE TABLE dbo.Stocks
(
  dt    DATETIME NOT NULL PRIMARY KEY,
  price INT      NOT NULL
);

INSERT INTO dbo.Stocks(dt, price) VALUES('20060801', 13);
```

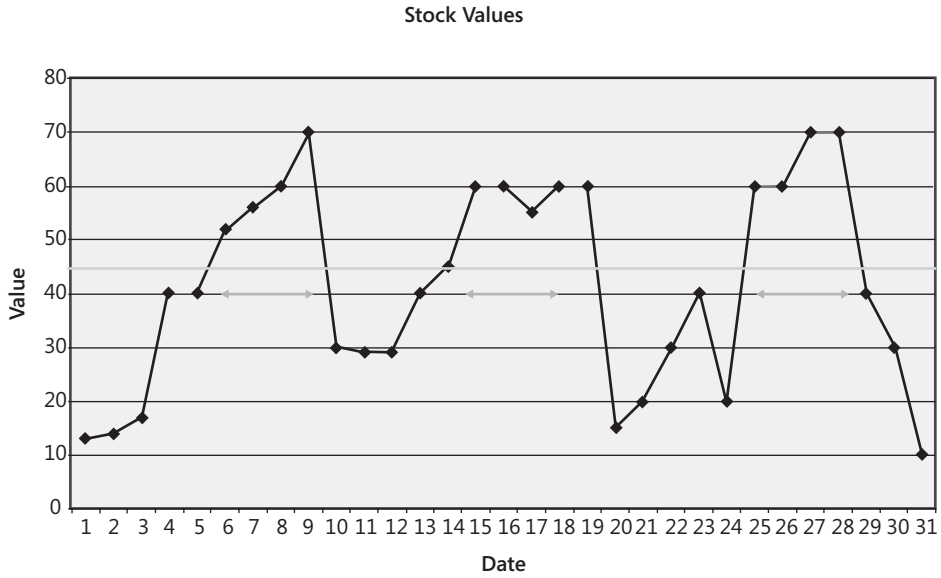
```
INSERT INTO dbo.Stocks(dt, price) VALUES('20060802', 14);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060803', 17);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060804', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060805', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060806', 52);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060807', 56);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060808', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060809', 70);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060810', 30);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060811’, 29);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060812', 29);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060813', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060814', 45);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060815', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060816', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060817', 55);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060818', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060819', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060820', 15);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060821', 20);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060822', 30);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060823', 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060824', 20);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060825', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060826', 60);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060827', 70);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060828', 70);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060829’, 40);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060830', 30);
INSERT INTO dbo.Stocks(dt, price) VALUES('20060831', 10);

CREATE UNIQUE INDEX idx_price_dt ON Stocks(price, dt);
```

טבלת Stocks מכילה מחירי מניות יומיים.

שים לב: מחירי מניות לרוב אינם מוגבלים למספרים שלמים, ובדרך כלל קיימת יותר ממניה אחת, אך אני אשתמש במספרים שלמים ובמניה אחת לצורך הפשטה. כמו כן, בשוקי מניות לרוב אין פעילות בימי שבת; מכיוון שברצוני להציג שיטה על פני רצף ללא פערים, הוספתי שורות גם לשבתות, עם ערך זהה לזה שאוחסן ליום שישי הקודם להם.

הבקשה היא לבודד תקופות עוקבות בהן מחיר המניה היה גבוה–שווה ל–50. תרשים 2–6 מציג את מחיר המניה לאורך זמן בצורה גרפית, והחיצים מייצגים את התקופות אותן עליך להחזיר.

תרשים 2–6: תקופות בהן ערכי מניות היו גבוהים יותר או שווים ל–50



לכל תקופה כזו, עליך להחזיר את תאריך ההתחלה, תאריך הסיום, משך בימים, ומחיר השיא (מקסימום).

הבה נתחיל בפתרון שאינו משתמש במספרי שורה. הצעד הראשון כאן הוא לסנן את השורות בהן המחיר גבוה או שווה ל–50. שלא כמו בבעיה המסורתית בה יש לך פערים ברצף, כאן הפערים מופיעים רק לאחר סינון. הרצף המלא עדיין מופיע בטבלת Stocks. תוכל להשתמש בעובדה זו לטובתך. כמובן, תוכל לקחת את המסלול הארוך לחישוב התאריך המקסימלי בתוך הקבוצה (התאריך הראשון שהוא גם מאוחר יותר או שווה לתאריך הנוכחי וגם שלאחריו יש פער). עם זאת, שיטה מהירה ופשוטה הרבה יותר לחישוב גורם ההקבצה תהיה להחזיר את התאריך הראשון הגדול יותר מהנוכחי, בו מחיר המניה נמוך מ–50. כאן אתה עדיין מקבל את אותו גורם הקבצה לכל האלמנטים של אותה קבוצת יעד, ועדיין אתה נזקק רק לרמת קינון אחת של תת–שאילתות במקום שתיים.

להלן השאילתה המייצרת את התוצאה הרצויה המוצגת בטבלה 6–31:

```
SELECT MIN(dt) AS startrange, MAX(dt) AS endrange,
  DATEDIFF(day, MIN(dt), MAX(dt)) + 1 AS numdays,
  MAX(price) AS maxprice
FROM (SELECT dt, price,
        (SELECT MIN(dt)
         FROM dbo.Stocks AS S2
         WHERE S2.dt > S1.dt
           AND price < 50) AS grp
      FROM dbo.Stocks AS S1
      WHERE price >= 50) AS D
GROUP BY grp;
```

טבלה 6–31: תחומים בהם ערכי המניה היו גדולים–שווים ל–50

startrange	*endrange*	*numdays*	*maxprice*
2006-08-06 00:00:00.000	2006-08-10 00:00:00.000	4	70
2006-08-15 00:00:00.000	2006-08-20 00:00:00.000	5	60
2006-08-25 00:00:00.000	2006-08-29 00:00:00.000	4	70

כמובן שב– SQL Server 2005 תוכל להשתמש בפונקציה ROW_NUMBER כפי שתיארתי בפרק 4:

```
SELECT MIN(dt) AS startrange, MAX(dt) AS endrange,
  DATEDIFF(day, MIN(dt), MAX(dt)) + 1 AS numdays,
  MAX(price) AS maxprice
FROM (SELECT dt, price,
        dt - ROW_NUMBER() OVER(ORDER BY dt) AS grp
      FROM dbo.Stocks AS S1
      WHERE price >= 50) AS D
GROUP BY grp;
```

CUBE ו–ROLLUP

CUBE ו–ROLLUP הן אופציות זמינות לשאילתות המכילות פסוקית GROUP BY. הן שימושיות עבור יישומים הנדרשים לספק מגוון משתנה של צבירת נתונים בהתבסס על סט משתנה של מאפיינים או ממדים (**dimensions**). (בהקשר של קוביות, משתמשים לעיתים קרובות במילה ממד, או כמילה נרדפת למאפיין או לתיאור תחום ערכים למאפיין). ראשית אתאר את האפשרות CUBE, ולאחר מכן אתאר את האפשרות ROLLUP, שהיא מקרה מיוחד של CUBE.

CUBE

תאר לך שהיישום שלך נדרש לספק למשתמשים את האפשרות לבקש פונקציות צבירה בהתבסס על סטים שונים של ממדים. נאמר, למשל, שנתוני הבסיס שלך הם טבלת Orders בה השתמשתי מוקדם יותר בפרק, ושהמשתמשים צריכים לנתח את הנתונים בהתבסס על שלושה ממדים: עובד, לקוח ושנת הזמנה. אם תקבץ את הנתונים לפי כל שלושת הממדים, תכסה רק את אחת האפשרויות בהן יהיו מעוניינים המשתמשים. עם זאת, המשתמשים עשויים לבקש כל סט ממדים (למשל, רק עובד, רק לקוח, רק שנת הזמנה, לקוח ועובד, וכך הלאה). לכל בקשה, תצטרך לבנות שאילתת GROUP BY שונה, לשלוח אותה ל– SQL Server ולהחזיר את התוצאה ללקוח. מדובר בהמון טיולים הלוך–חזור והמון תנועה ברשת.

ככל שמספר הממדים גדל, מספר שאילתות ה– GROUP BY האפשריות גדל משמעותית. עבור n ממדים, קיימות 2^n שאילתות שונות. עם 3 ממדים, אתה עומד בפני 8 בקשות אפשריות; עם 4 ממדים, קיימות 16. עם 10 ממדים (המספר המקסימלי של ביטויי הקבצה בו נוכל להשתמש עם CUBE), משתמשים יכולים לבקש כל אחת מ–1024 שאילתות GROUP BY שונות.

במילים פשוטות, הוספת האפשרות WITH CUBE לשאילתה, עם כל הממדים המוגדרים בפסוקית GROUP BY מייצרת סט אחד מאוחד מתוך כל הסטים של תוצאה של כל שאילתות ה– GROUP BY השונות על פני תת–סטים של הממדים. אם תחשוב על כך, קוביות Analysis Services נותנות לך שימושיות דומה, אך בקנה מידה גדול הרבה יותר ועם אופציות מתוחכמות יותר משמעותית. עם זאת, כאשר אינך צריך לתמוך בניתוח דינמי בקנה מידה כזה וברמה כזו של תחכום, האפשרות WITH CUBE מאפשרת לך להשיג זאת בתוך מסד הנתונים הרלציוני.

מכיוון שכל סט ממדים מייצר סט תוצאה עם תת–סט שונה של כל טורי התוצאה האפשריים, המעצבים שהטמיעו CUBE ו–ROLLUP היו צריכים למצוא שומר–מקום לערכים בטורים הלא–נדרשים. המעצבים בחרו ב–NULL. כך, למשל, לכל השורות מסט התוצאה של GROUP BY empid, custid יהיה NULL בטור התוצאה orderyear. דבר זה מאפשר לכל הסטים של התוצאה להיות מאוחדים לסט תוצאה אחד עם סכמה אחת.

כדוגמה, שאילתת ה–CUBE הבאה מחזירה את כל הצבירות האפשריות (כמויות כוללות) של הזמנות בהתבסס על הממדים empid, custid ו–orderyear, ומפיקה את הפלט המוצג בטבלה 32–6:

```
SELECT empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY empid, custid, YEAR(orderdate)
WITH CUBE;
```

טבלה 32–6: תוצאת CUBE

empid	*custid*	*orderyear*	*totalqty*
1	A	2002	12
1	A	NULL	12
1	B	2002	20
1	B	NULL	20
1	C	2003	14
1	C	NULL	14
1	NULL	NULL	46
2	B	2003	12
2	B	NULL	12
2	C	2004	20
2	C	NULL	20
2	NULL	NULL	32
3	A	2002	10
3	A	NULL	10
3	B	2004	15
3	B	NULL	15
3	C	2002	22
3	C	NULL	22
3	D	2002	30
3	D	NULL	30
3	NULL	NULL	77

empid	*custid*	*orderyear*	*totalqty*
4	A	2003	40
4	A	2004	10
4	A	NULL	50
4	NULL	NULL	50
NULL	NULL	NULL	205
NULL	A	2002	22
NULL	A	2003	40
NULL	A	2004	10
NULL	A	NULL	72
NULL	B	2002	20
NULL	B	2003	12
NULL	B	2004	15
NULL	B	NULL	47
NULL	C	2002	22
NULL	C	2003	14
NULL	C	2004	20
NULL	C	NULL	56
NULL	D	2002	30
NULL	D	NULL	30
1	NULL	2002	32
3	NULL	2002	62
NULL	NULL	2002	94
1	NULL	2003	14
2	NULL	2003	12
4	NULL	2003	40
NULL	NULL	2003	66
2	NULL	2004	20
3	NULL	2004	15
4	NULL	2004	10
NULL	NULL	2004	45

כל עוד טורי הממדים בטבלה אינם מכילים NULLs, בכל מקום בו אתה רואה NULL בתוצאה של שאילתת ה–CUBE, המשמעות הלוגית היא הכל. מאוחר יותר אדון כיצד לטפל ב–NULLs בטבלה עליה מבוצעת השאילתה. למשל, השורה המכילה NULL, NULL, 2004, 45 מראה את הכמות הכוללת (45) עבור ההזמנות של כל העובדים וכל הלקוחות בשנת הזמנה 2004. ייתכן שתרצה להעלות את סט התוצאה משאילתת ה–CUBE ל–cache אצל הלקוח או בשכבת ביניים, או שתרצה לשמור אותו בטבלה זמנית ולבנות לה אינדקס. הקוד בקטע–קוד 6–15 בוחר את סט התוצאה לטבלה הזמנית #Cube ואז יוצר אינדקס–clustered על כל הממדים.

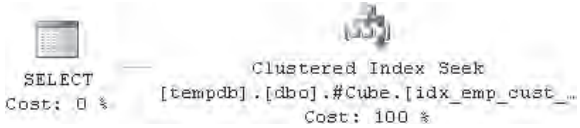
קטע–קוד 6–15: מילוי #Cube עם סט התוצאה של שאילתת CUBE

```
SELECT empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
INTO #Cube
FROM dbo.Orders
GROUP BY empid, custid, YEAR(orderdate)
WITH CUBE;

CREATE CLUSTERED INDEX idx_emp_cust_year
  ON #Cube(empid, custid, orderyear);
```

ניתן לספק כל בקשה לצבירה על ידי פעולת חיפוש בתוך האינדקס–clustered. למשל, השאילתה הבאה מחזירה את הכמות הכוללת לעובד 1 ומפיקה את תוכנית העבודה המוצגת בתרשים 6–3:

```
SELECT totalqty
FROM #Cube
WHERE empid = 1
  AND custid IS NULL
  AND orderyear IS NULL;
```

תרשים 6–3: תוכנית עבודה לשאילתה על טבלת #Cube



כשתסיים לבצע שאילתות על טבלת #Cube, הסר אותה:

```
DROP TABLE #Cube;
```

במקרה שטורי ממד מאפשרים NULLs, עשויה להתעורר סוגיה נוספת. למשל, הרץ את הקוד הבא כדי לאפשר NULLs בטור empid והוסף מספר ערכי NULL:

```
ALTER TABLE dbo.Orders ALTER COLUMN empid INT NULL;
UPDATE dbo.Orders SET empid = NULL WHERE orderid IN(10001, 20001);
```

עליך להבין שכאשר אתה מריץ שאילתת CUBE כעת, NULL בטור empid אינו חד–משמעי. כאשר הוא נוצר מ–NULL בטור empid, הוא מייצג קבוצה של עובדים לא ידועים. כאשר הוא מיוצר על ידי האפשרות CUBE, הוא מייצג את כל העובדים. ללא כל טיפול מיוחד ב–NULLs, לא תוכל לדעת מה המשמעות. אני אוהב להחליף פשוט את NULL בערך שאני יודע שאינו יכול להופיע בנתונים – למשל, 1- כ–empid. אני משתמש בפונקציות COALESCE או ISNULL למטרה זו. לאחר החלפה זו, הערך 1- ייצג עובדים לא ידועים, ול–NULL יכולה להיות רק המשמעות של כל העובדים. להלן השאילתה המיישמת היגיון זה:

```
SELECT COALESCE(empid, -1) AS empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY COALESCE(empid, -1), custid, YEAR(orderdate)
WITH CUBE;
```

אפשרות נוספת היא להשתמש בפונקציית T-SQL: GROUPING, שעוצבה לטפל ברב–משמעיות של NULL בסט התוצאה. אתה מספק לפונקציה את שם טור הממד כקלט. הערך של GROUPING(<dimension>) מציין האם הערך של <dimension> בשורה מייצג קבוצה מסוימת (במקרה זה, GROUPING מחזיר 0), או שהוא שומר מקום המייצג את כל הערכים (במקרה זה, GROUPING מחזיר 1). בפרט עבור ערך הממד NULL, GROUPING מחזיר 1 אם ה–NULL הוא תוצאה של האפשרות CUBE (כלומר הכל) ו–0 אם הוא מייצג את הקבוצה של ה–NULLs המקוריים. להלן שאילתה המשתמשת בפונקציה GROUPING:

```
SELECT empid, GROUPING(empid) AS grp_empid, custid,
  YEAR(orderdate) AS orderyear, SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY empid, custid, YEAR(orderdate)
WITH CUBE;
```

אם אתה מכניס את סט התוצאה של שאילתת CUBE לטבלה זמנית, אל תשכח לכלול את טורי ההקבצה באינדקס, וכן ודא שאתה כולל אותם במסננים שלך. למשל, הנח שהכנסת את סט התוצאה של השאילתה הקודמת לטבלה זמנית בשם #Cube. השאילתה הבאה תחזיר את הכמות הכוללת ללקוח A:

```
SELECT totalqty
FROM #Cube
WHERE empid IS NULL AND grp_empid = 1
  AND custid = 'A'
  AND orderyear IS NULL;
```

ROLLUP

ROLLUP הוא מקרה פרטי של CUBE בו ניתן להשתמש כאשר הממדים יוצרים היררכיה. למשל, נניח שברצונך לנתח כמויות הזמנה בהתבסס על הממדים שנת הזמנה, חודש הזמנה ויום הזמנה. הנח שלא חשובות לך הכמויות הכוללות של פריט ברמת גרעיניות אחת עבור כל הערכים ברמה גבוהה יותר של גרעיניות – למשל, הכמויות הכוללות של היום השלישי בכל החודשים ובכל השנים. חשובות לך רק הכמויות הכוללות של פריט ברמה אחת של גרעיניות לכל ערכי הרמות הנמוכות יותר – למשל, הכמות הכוללת לשנת 2004, כל החודשים, כל הימים. ROLLUP מאפשרת לך להשיג את מטרתך. היא נפטרת מכל פונקציות הצבירה ה"לא מעניינות" במקרה של היררכיה. באופן מדויק יותר, היא אינה טורחת אפילו לחשב אותן, וכך תוכל לצפות לביצועים טובים יותר משאילתת ROLLUP מאשר משאילתת CUBE המבוססת על אותם ממדים.

כדוגמה לשימוש ב–ROLLUP, השאילתה הבאה מחזירה את כמויות ההזמנה הכוללות לממדים שנת הזמנה, חודש הזמנה, ויום הזמנה, והיא מייצרת את הפלט המוצג בטבלה 6–33:

```
SELECT
  YEAR(orderdate)    AS orderyear,
  MONTH(orderdate)   AS ordermonth,
  DAY(orderdate)     AS orderday,
  SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY YEAR(orderdate), MONTH(orderdate), DAY(orderdate)
WITH ROLLUP;
```

טבלה 6–33: תוצאת ROLLUP

orderyear	*ordermonth*	*orderday*	*totalqty*
2002	4	18	22
2002	4	NULL	22
2002	8	2	10
2002	8	NULL	10
2002	9	7	30
2002	9	NULL	30
2002	12	24	32
2002	12	NULL	32
2002	NULL	NULL	94
2003	1	9	40
2003	1	18	14
2003	1	NULL	54
2003	2	12	12
2003	2	NULL	12
2003	NULL	NULL	66
2004	2	12	10
2004	2	16	20
2004	2	NULL	30
2004	4	18	15
2004	4	NULL	15
2004	NULL	NULL	45
NULL	NULL	NULL	205

סיכום

פרק זה עסק בפתרונות שונים לבעיות פונקציות צבירה (אגרגציות) של נתונים שהשתמשו בשיטות מפתח לביצוע שאילתות שהצגתי בפרקים קודמים. הוא הציג גם שיטות חדשות, כמו טיפול בשוברי–שוויון על ידי שימוש בשרשור, חישוב מינימום על ידי שימוש בפונקציה MAX, סיבוב על ציר, חישוב פונקציות צבירה מוגדרות–משתמש על ידי שימוש בשיטות ייחודיות ואחרות.

כפי שבוודאי הבחנת, שיטות צבירה של נתונים משלבות מניפולציות לוגיות רבות. אם אתה מחפש דרכים לשיפור הלוגיקה שלך, תוכל לתרגל חידות היגיון טהורות, שכן יש להן הרבה במשותף עם בעיות שאילתות מבחינת התהליך החשיבתי המעורב. תוכל למצוא חידות היגיון טהורות בנספח א'.

7

TOP ו–APPLY

פרק זה מציג שני אלמנטים של שאילתות שלכאורה נראה שאין ביניהם קשר. אלמנט אחד היא האפשרות TOP, המאפשרת לך להגביל את מספר השורות המושפעות מהשאילתה. האלמנט השני הוא האופרטור הטבלאי החדש APPLY, המאפשר לך להפעיל ביטוי טבלה על כל שורה של ביטוי טבלה אחר – ולמעשה ליצור join מקושר. החלטתי לדון באלמנטים אלה באותו פרק, מכיוון שאני מוצא שלעיתים קרובות למדי ניתן להשתמש בהם ביחד לפתרון בעיות שאילתה.

תחילה אתאר את העקרונות של TOP ושל APPLY, ואז אמשיך בפתרונות לבעיות נפוצות המשתמשות באלמנטים אלו.

SELECT TOP

בשאילתת SELECT או בביטוי טבלה, TOP משמש בצירוף פסוקית ORDER BY, כדי להגביל את שורות התוצאה המופיעות ראשונות בסדר של ה– ORDER BY. באפשרותך לציין את כמות השורות בה אתה מעוניין באחת משתי דרכים: כמספר מדויק של שורות, מ–TOP(0) ועד TOP(9223372036854775807) (הערך BIGINT הגבוה ביותר), או כאחוז שורות, מ–TOP(0E0) ועד TOP(100E0), על ידי שימוש בערך FLOAT. ב– Microsoft SQL Server 2000 ניתן להשתמש אך ורק בקבוע כדי לציין את ההגבלה שלך. SQL Server 2005 תומך בכל ביטוי עצמאי, ולא רק בקבועים, עם TOP.

בשביל להבהיר אילו שורות הן שורות ה–"top" המושפעות משאילתת TOP, עליך לציין סדר של שורות. ממש כפי שאינך יכול להבדיל בין ראש לבין תחתית אלא אם כן ידוע לך מהו הכיוון כלפי מעלה, לא תוכל לדעת אילו שורות מושפעות מ–TOP אלא אם אתה מציין פסוקית ORDER BY. עליך לחשוב על TOP ועל ORDER BY ביחד כמסנן לוגי ופחות כמנגנון מיון. זאת הסיבה ששאילתה המכילה הן TOP והן פסוקית ORDER BY מחזירה טבלה מוגדרת–היטב ומותרת בביטויי טבלה. בשאילתה ללא TOP, הפסוקית ORDER BY ממלאת תפקיד אחר – היא מגדירה את הסדר שבו יוחזרו התוצאות. שימוש ב– ORDER BY ללא TOP אינו מותר בביטויי טבלה.

שים לב: מעניין שבאפשרותך לציין אפשרות TOP בשאילתה ללא פסוקית ORDER BY, אך המשמעות הלוגית של TOP בשאילתה כזו אינה מוגדרת לחלוטין. אסביר על היבט זה של TOP מייד.

הבה נתחיל בדוגמה פשוטה. השאילתה הבאה מחזירה את שלוש ההזמנות האחרונות, ומפיקה את הקלט המוצג בטבלה 1–7:

```
USE Northwind;
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC;
```

טבלה 1–7: שלוש הזמנות אחרונות

OrderID	*CustomerID*	*OrderDate*
11077	RATTC	1998-05-06 00:00:00.000
11076	BONAP	1998-05-06 00:00:00.000
11075	RICSU	1998-05-06 00:00:00.000

מיון ראשית לפי OrderDate DESC מבטיח שתקבל את ההזמנות האחרונות ביותר. מכיוון ש–OrderDate אינו ייחודי, הוספתי את OrderID DESC לרשימת ה– ORDER BY כשובר–שוויון. מבין הזמנות בעלות אותו OrderDate, שובר–השוויון ייתן קדימות להזמנות עם ערך OrderID גבוה יותר.

שים לב: השתמשתי כאן בסוגריים עבור ביטוי הקלט לאפשרות ה–TOP. מכיוון ש– SQL Server 2005 תומך בכל ביטוי עצמאי כקלט, הביטוי חייב להירשם בתוך סוגריים. מסיבות של תאימות–אחורה, SQL Server 2005 עדיין תומך בשאילתות SELECT TOP המשתמשות בקבוע ללא סוגריים. עם זאת, זהו הרגל טוב לשים קבועים של TOP בתוך סוגריים כדי להסתגל לדרישות החדשות.

כדוגמה לאפשרות PERCENT, השאילתה הבאה מחזירה את ההזמנות האחרונות שבוצעו, המהוות אחוז אחד מכלל ההזמנות, ומפיקה את הפלט המוצג בטבלה 2–7:

```
SELECT TOP(1) PERCENT OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC;
```

טבלה 7–2: הזמנות אחרונות המהוות אחוז מכלל ההזמנות

OrderID	*CustomerID*	*OrderDate*
11077	RATTC	1998-05-06 00:00:00.000
11076	BONAP	1998-05-06 00:00:00.000
11075	RICSU	1998-05-06 00:00:00.000
11074	SIMOB	1998-05-06 00:00:00.000
11073	PERIC	1998-05-05 00:00:00.000
11072	ERNSH	1998-05-05 00:00:00.000
11071	LILAS	1998-05-05 00:00:00.000
11070	LEHMS	1998-05-05 00:00:00.000
11069	TORTU	1998-05-04 00:00:00.000

טבלת Orders מכילה 830 שורות, ואחוז אחד מ–830 הוא 8.3. מכיוון שרק שורות שלמות יכולות להיות מוחזרות, והתבקשו 8.3, המספר בפועל של שורות שהוחזרו הוא 9. כאשר משתמשים ב– TOP ...PERCENT, והאחוז המצוין כולל שורה חלקית, מספר השורות המדויק שהתבקש מעוגל כלפי מעלה.

TOP ודטרמיניזם

כפי שהזכרתי קודם, שאילתת TOP אינה דורשת פסוקית ORDER BY. עם זאת, שאילתה כזו אינה דטרמיניסטית. כלומר, הרצת השאילתה פעמיים מול אותם נתונים עשויה להפיק סטי תוצאה שונים, ושניהם יהיו נכונים. השאילתה הבאה מחזירה שלוש הזמנות, כאשר אין כלל הקובע אילו שלוש יחזרו:

```
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders;
```

כאשר הרצתי שאילתה זו, קיבלתי את הפלט המוצג בטבלה 7–3, אך אתה עשוי לקבל פלט שונה. SQL Server יחזיר את שלוש השורות הראשונות שאליהן הוא ניגש במקרה.

טבלה 7–3: תוצאה של שאילתת TOP ללא ORDER BY

OrderID	*CustomerID*	*OrderDate*
10248	VINET	1996-07-04 00:00:00.000
10249	TOMSP	1996-07-05 00:00:00.000
10250	HANAR	1996-07-08 00:00:00.000

שים לב: אני יכול לחשוב רק על שתי סיבות טובות לשימוש ב– SELECT TOP ללא ORDER BY, ואני לא ממליץ על שימוש באפשרות זו בשום מקרה אחר. סיבה אחת לשימוש ב– SELECT TOP היא עבור תזכורת מהירה של מבנה טבלה או של שמות טורים בטבלה, או כדי לראות האם הטבלה מכילה נתונים כלשהם. הסיבה השנייה לשימוש ב– SELECT TOP – ובפרט ב– SELECT TOP(0) – היא כדי ליצור טבלה ריקה עם מבנה זהה לטבלה או לשאילתה אחרת. במקרה זה, תוכל להשתמש ב– SELECT TOP(0) <column list> INTO <table name> FROM…. כמובן, אינך זקוק לפסוקית ORDER BY כדי לציין "אילו אפס שורות" ברצונך לבחור!

שאילתת TOP יכולה להיות לא–דטרמיניסטית אפילו כאשר פסוקית ORDER BY מוגדרת, אם רשימת ה– ORDER BY אינה ייחודית. למשל, השאילתה הבאה מחזירה את שלוש ההזמנות הראשונות בסדר CustomerID עולה, ומפיקה את הפלט המוצג בטבלה 4–7:

```
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY CustomerID;
```

טבלה 4–7: תוצאת שאילתת TOP עם רשימת ORDER BY לא ייחודית

OrderID	*CustomerID*	*OrderDate*
10643	ALFKI	1997-08-25 00:00:00.000
10692	ALFKI	1997-10-03 00:00:00.000
10702	ALFKI	1997-10-13 00:00:00.000

מובטח לך שתקבל את ההזמנות עם ערכי CustomerID הנמוכים ביותר. עם זאת, מכיוון שהטור CustomerID אינו ייחודי, אינך יכול להבטיח אילו שורות מבין אלו עם אותם ערכי CustomerID יוחזרו. שוב, תקבל את אלו אליהן SQL Server ניגש במקרה ראשונות. דרך אחת להבטיח דטרמיניזם היא להוסיף שובר–שוויון אשר גורם לרשימת ה– ORDER BY להיות ייחודית – למשל, המפתח הראשי:

```
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY CustomerID, OrderID;
```

דרך אחרת להבטיח דטרמיניזם היא להשתמש באפשרות WITH TIES. כאשר אתה משתמש ב– WITH TIES, השאילתה תייצר סט תוצאה הכולל את כל השורות הנוספות, אשר הערכים שלהן בטור או בטורי המיון שווים לאלו שבשורה האחרונה שהוחזרה. למשל, השאילתה

הבאה מציינת TOP(3), ועדיין מחזירה שבע שורות המוצגות בטבלה 5–7. ארבע שורות נוספות מוחזרות מכיוון שיש להן אותו ערך CustomerID (ALFKI) כמו לשורה השלישית:

```
SELECT TOP(3) WITH TIES OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY CustomerID;
```

טבלה 5–7: תוצאה של שאילתת TOP המשתמשת באפשרות WITH TIES

OrderID	*CustomerID*	*OrderDate*
10643	ALFKI	1997-08-25 00:00:00.000
10692	ALFKI	1997-10-03 00:00:00.000
10702	ALFKI	1997-10-13 00:00:00.000
10835	ALFKI	1998-01-15 00:00:00.000
10952	ALFKI	1998-03-16 00:00:00.000
11011	ALFKI	1998-04-09 00:00:00.000
10643	ALFKI	1997-08-25 00:00:00.000

שים לב: ישנם יישומים שחייבים להבטיח דטרמיניזם. למשל, אם אתה משתמש באפשרות TOP לצורך דפדוף, אינך מעוניין שאותה שורה תופיע בשני דפים עוקבים רק מכיוון שהשאילתה לא הייתה דטרמיניסטית. זכור שאתה תמיד יכול להוסיף את המפתח הראשי כשובר–שוויון כדי להבטיח דטרמיניזם, במקרה שרשימת ה– ORDER BY אינה ייחודית.

TOP וביטויי קלט

כקלט ל– TOP, SQL Server 2005 תומך בכל ביטוי עצמאי המניב תוצאה סקלארית. ניתן להשתמש בביטוי עצמאי מהשאילתה החיצונית – משתנה, ביטוי אריתמטי, או אפילו תוצאה של תת–שאילתה. למשל, השאילתה הבאה מחזירה @n הזמנות אחרונות, כאשר @n הוא משתנה:

```
DECLARE @n AS INT;
SET @n = 2;

SELECT TOP(@n) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC;
```

השאילתה הבאה מציגה את השימוש בתת–שאילתה כקלט ל–TOP. כמו תמיד, הקלט ל–TOP מציין את מספר השורות שהשאילתה מחזירה – עבור דוגמה זו, מספר השורות המוחזרות הוא מספר ההזמנות החודשי הממוצע. הפסוקית ORDER BY בדוגמה זו מציינת שהשורות המוחזרות הן האחרונות, כאשר OrderID הוא שובר–השוויון (ID גבוה מנצח):

```
SELECT TOP(SELECT COUNT(*)/(DATEDIFF(month,
             MIN(OrderDate), MAX(OrderDate))+1)
           FROM dbo.Orders)
  OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC;
```

מספר ההזמנות החודשי הממוצע הוא מספר ההזמנות הכולל מחולק ב– [ההפרש בחודשים בין תאריך ההזמנה הגבוה ביותר לבין הנמוך ביותר ועוד אחד]. מכיוון שישנן 830 הזמנות בטבלה שבוצעו בתקופה בת 23 חודשים, הפלט מכיל את 36 ההזמנות האחרונות.

TOP ושינויי נתונים

SQL Server 2005 מספק אפשרות TOP עבור משפטי שינוי נתונים (INSERT, UPDATE ו–DELETE).

שים לב: טרם SQL Server 2005, האפשרות SET ROWCOUNT סיפקה יכולת דומה לחלק מהיכולות החדשות של TOP. SET ROWCOUNT קיבלה משתנה כקלט, והשפיעה הן על משפטי שינוי נתונים והן על משפטי SELECT. מיקרוסופט לא ממליצה יותר על SET ROWCOUNT כדרך להשפיע על משפטי INSERT, UPDATE ו–DELETE – למעשה, ב–Katmai, הגרסה הבאה של SQL Server, SET ROWCOUNT לא תשפיע יותר כלל על משפטי שינוי נתונים. השתמש ב–TOP כדי להגביל את מספר השורות המושפעות על ידי משפטי שינוי נתונים.

SQL Server 2005 מציג תמיכה לאפשרות TOP עם משפטי שינוי נתונים, המאפשרת לך להגביל את מספר או אחוז השורות המושפעות. ציון TOP יכול להתווסף לאחר מילות המפתח DELETE, UPDATE או INSERT.

פסוקית ORDER BY אינה נתמכת עם משפטי שינוי נתונים, אפילו כאשר משתמשים ב–TOP, כך שאף אחד ממשפטי שינוי הנתונים אינו יכול להסתמך על מיון. SQL Server פשוט ישפיע על המספר המבוקש של שורות ראשונות אליהן הוא ניגש במקרה.

במשפט הבא, SQL Server אינו מבטיח אילו שורות יתווספו משורות המקור:

```
INSERT TOP(10) INTO target_table
  SELECT col1, col2, col3
  FROM source_table;
```

שים לב: על אף שאינך יכול להשתמש ב– ORDER BY עם INSERT TOP, תוכל להבטיח אילו שורות ייכנסו אם תגדיר TOP ו– ORDER BY במשפט ה–SELECT כלהלן:

```
INSERT INTO target_table
  SELECT TOP(10) col1, col2, col3
  FROM source_table
  ORDER BY col1;
```

INSERT TOP שימושי כאשר ברצונך לטעון תת–סט של שורות מטבלה גדולה או מסט תוצאה גדול ולא חשוב לך איזה תת–סט ייבחר, אלא רק מספר השורות.

שים לב: על אף ש– ORDER BY אינו ניתן לשימוש עם UPDATE TOP ו– DELETE TOP, תוכל להתגבר על המגבלה על ידי יצירת CTE משאילתת SELECT TOP, אשר יש לה פסוקית ORDER BY, ואז לבצע את ה–UPDATE או ה–DELETE שלך מול ה–CTE:

```
WITH CTE_DEL AS
(
  SELECT TOP(10) * FROM some_table ORDER BY col1
)
DELETE FROM CTE_DEL;

WITH CTE_UPD AS
(
  SELECT TOP(10) * FROM some_table ORDER BY col1
)
UPDATE CTE_UPD SET col2 = col2 + 1;
```

ניתן לראות דוגמה למצב כזה כאשר עליך להכניס או לשנות נפחי נתונים גדולים ומסיבות מעשיות, אתה מחלק אותם למקטעים, ומשנה תת–סט אחד בכל פעם. למשל, ניקוי נתונים היסטוריים עשוי לערב מחיקה של מיליוני שורות נתונים. אלא אם כן טבלת היעד מחולקת למחיצות ואתה יכול פשוט להסיר מחיצות, תהליך הניקוי דורש משפט DELETE. למחיקת סט כה גדול של שורות בטרנזקציה בודדת, קיימים מספר חסרונות. משפט DELETE נרשם בלוג בצורה מלאה, והוא ידרוש מספיק מקום בלוג הטרנזקציות כדי להכיל את הטרנזקציה כולה. במהלך פעולת מחיקה, שעשויה לארוך זמן רב, לא ניתן לדרוס אף

חלק מהלוג, מהטרנזקציה הפתוחה הישנה ביותר ועד לנקודה הנוכחית. יתרה מכך, אם הטרנזקציה נפסקת באמצע מסיבה כלשהי, כל הפעילות שהתרחשה עד לנקודה זו תעבור roll back, ודבר זה יארך לא מעט זמן. לבסוף, כאשר הרבה מאוד שורות נמחקות בבת אחת, SQL Server עלול להסלים את הנעילות האינדיבידואליות המוחזקות על השורות הנמחקות לכדי נעילה בלעדית על כל הטבלה, מה שימנע הן קריאה והן כתיבה לטבלת היעד עד שה–DELETE מסתיים.

הגיוני לפצל את טרנזקציית ה–DELETE הבודדת לכמה קטנות יותר – קטנות מספיק כדי לא לגרום להסלמת נעילה (לרוב, מספר אלפי שורות בכל טרנזקציה), ולאפשר מיחזור של לוג הטרנזקציות. תוכל לוודא בקלות שהמספר שבחרת לא גורם להסלמת נעילה על ידי בדיקה של DELETE עם האפשרות TOP, בעוד אתה מנטר אירועי Lock Escalation בעזרת Profiler. פיצול טרנזקציית ה–DELETE הגדולה, יאפשר גם דריסה של החלק הלא–פעיל של הלוג שעבר כבר גיבוי.

כדי להדגים ניקוי נתונים במספר טרנזקציות, הרץ את הקוד הבא, היוצר את טבלת LargeOrders וממלא אותה בנתונים לדוגמה:

```
IF OBJECT_ID('dbo.LargeOrders') IS NOT NULL
  DROP TABLE dbo.LargeOrders;
GO
SELECT IDENTITY(int, 1, 1) AS OrderID,
  O1.CustomerID, O1.EmployeeID, O1.OrderDate, O1.RequiredDate,
  O1.ShippedDate, O1.ShipVia, O1.Freight, O1.ShipName, O1.ShipAddress,
  O1.ShipCity, O1.ShipRegion, O1.ShipPostalCode, O1.ShipCountry
INTO dbo.LargeOrders
FROM dbo.Orders AS O1, dbo.Orders AS O2;

CREATE UNIQUE CLUSTERED INDEX idx_od_oid
  ON dbo.LargeOrders(OrderDate, OrderID);
```

בגרסאות קודמות ל– SQL Server 2005, השתמש באפשרות SET ROWCOUNT כדי לפצל DELETE גדול, כפי שמציג הפתרון הבא:

```
SET ROWCOUNT 5000;
WHILE 1 = 1
BEGIN
  DELETE FROM dbo.LargeOrders
  WHERE OrderDate < '19970101';

  IF @@rowcount < 5000 BREAK;
END
SET ROWCOUNT 0;
```

הקוד קובע את האפשרות ROWCOUNT על 5000, ובכך מגביל את מספר השורות המושפעות על ידי כל DML ל–5000. לולאה אינסופית מנסה למחוק 5000 שורות בכל חזרה, כאשר כל מחיקה של 5000 שורות מתרחשת בטרנזקציה נפרדת. הלולאה נשברת ברגע שהמקטע האחרון מטופל (כלומר, כאשר מספר השורות המושפעות קטן מ–5000). ב– SQL Server 2005 אינך זקוק יותר לאפשרות SET ROWCOUNT. פשוט ציין DELETE TOP(5000):

```
WHILE 1 = 1
BEGIN
  DELETE TOP(5000) FROM dbo.LargeOrders
  WHERE OrderDate < '19970101';

  IF @@rowcount < 5000 BREAK;
END
```

באופן דומה, תוכל לפצל עדכונים גדולים למקטעים. למשל, נאמר שאתה צריך לשנות את קוד הלקוח OLDWO ל–ABCDE, בכל מקום שהוא מופיע בטבלה LargeOrders. להלן הפתרון בו היית משתמש ב– SQL Server 2000 כאשר אתה נשען על האפשרות SET ROWCOUNT:

```
SET ROWCOUNT 5000;
WHILE 1 = 1
BEGIN
   UPDATE dbo.LargeOrders
     SET CustomerID = N'ABCDE'
   WHERE CustomerID = N'OLDWO';

   IF @@rowcount < 5000 BREAK;
END
SET ROWCOUNT 0;
```

ולהלן הפתרון ב– SQL Server 2005 המשתמש ב– UPDATE TOP:

```
WHILE 1 = 1
BEGIN
  UPDATE TOP(5000) dbo.LargeOrders
    SET CustomerID = N'ABCDE'
  WHERE CustomerID = N'OLDWO';

  IF @@rowcount < 5000 BREAK;
END
```

כאשר תסיים להתנסות בשינויי המקטעים, הסר את טבלת LargeOrders:

```
IF OBJECT_ID('dbo.LargeOrders') IS NOT NULL
  DROP TABLE dbo.LargeOrders;
```

APPLY

אופרטור הטבלה החדש APPLY מפעיל את ביטוי הטבלה הימני על כל שורה מביטוי הטבלה השמאלי. שלא כמו join, בו אין חשיבות לסדר בו כל אחד מביטויי הטבלה מוערך, APPLY חייב לוגית להעריך את ביטוי הטבלה השמאלי ראשון. סדר הערכה לוגי זה של הקלטים, מאפשר לביטוי הטבלה הימני להתייחס לשמאלי – דבר שהיה בלתי אפשרי קודם ל– SQL Server 2005. הרעיון יכול להיות ברור יותר על ידי דוגמה.

הרץ את הקוד הבא כדי ליצור פונקציה טבלאית פנימית הנקראת fn_top_products:

```
IF OBJECT_ID('dbo.fn_top_products') IS NOT NULL
  DROP FUNCTION dbo.fn_top_products;
GO
CREATE FUNCTION dbo.fn_top_products
  (@supid AS INT, @catid INT, @n AS INT)
  RETURNS TABLE
AS
RETURN
  SELECT TOP(@n) WITH TIES ProductID, ProductName, UnitPrice
  FROM dbo.Products
  WHERE SupplierID = @supid
    AND CategoryID = @catid
  ORDER BY UnitPrice DESC;
GO
```

הפונקציה מקבלת שלושה קלטים: קוד ספק (@supid), קוד קטגוריה (@catid) ומספר מבוקש של מוצרים (@n). הפונקציה מחזירה את מספר המוצרים המבוקש בקטגוריה הנתונה, שסופקו על ידי הספק הנתון, עם מחירי היחידה הגבוהים ביותר. השאילתה משתמשת באפשרות TOP עם WITH TIES כדי להבטיח סט תוצאה דטרמיניסטי על ידי הכללת כל המוצרים בעלי מחיר יחידה זהה לזה של המוצר הכי פחות יקר שהוחזר.

השאילתה הבאה משתמשת באופרטור APPLY יחד עם fn_top_products כדי להחזיר, לכל ספק, את שני המשקאות היקרים ביותר. קוד הקטגוריה למשקאות הוא 1, כך ש–1 מסופק עבור הפרמטר @catid. שאילתה זו מייצרת את הפלט המוצג בטבלה 6–7:

```
SELECT S.SupplierID, CompanyName, ProductID, ProductName, UnitPrice
FROM dbo.Suppliers AS S
  CROSS APPLY dbo.fn_top_products(S.SupplierID, 1, 2) AS P;
```

טבלה 6–7: שני המשקאות היקרים ביותר לכל ספק

SupplierID	*CompanyName*	*ProductID*	*ProductName*	*UnitPrice*
18	Aux joyeux ecclésiastiques	38	Côte de Blaye	263.50
18	Aux joyeux ecclésiastiques	39	Chartreuse verte	18.00
16	Bigfoot Breweries	35	Steeleye Stout	18.00
16	Bigfoot Breweries	67	Laughing Lumberjack Lager	14.00
16	Bigfoot Breweries	34	Sasquatch Ale	14.00
1	Exotic Liquids	2	Chang	19.00
1	Exotic Liquids	1	Chai	18.00
23	Karkki Oy	76	Lakkalikööri	18.00
20	Leka Trading	43	Ipoh Coffee	46.00
7	Pavlova, Ltd.	70	Outback Lager	15.00
12	Plutzer Lebensmittelgroßmärkte AG	75	Rhönbräu Klosterbier	7.75
10	Refrescos Americanas LTDA	24	Guaraná Fantástica	4.50

ציון CROSS עם האופרטור APPLY משמעותו שלא תוחזרנה שורות בסט התוצאה עבור שורה בביטוי הטבלה השמאלי, שעבורה ביטוי הטבלה הימני dbo.fn_top_products(S.SupplierID, 1, 2) הוא ריק. זהו המקרה כאן, למשל, עבור ספקים שאינם מספקים משקאות. כדי לכלול תוצאות גם מספקים אלו, השתמש במילת המפתח OUTER במקום ב–CROSS, כפי שמציגה השאילתה הבאה:

```
SELECT S.SupplierID, CompanyName, ProductID, ProductName, UnitPrice
FROM dbo.Suppliers AS S
  OUTER APPLY dbo.fn_top_products(S.SupplierID, 1, 2) AS P;
```

שאילתה זו מחזירה 33 שורות. סט התוצאה עם OUTER APPLY כולל שורות שמאליות שעבורן ביטוי הטבלה הימני הניב סט ריק, ועבור שורות אלה טורי ביטוי הטבלה הימני הם NULL.

קיימת תופעת לוואי נחמדה שנגרמה מהטכנולוגיה, שנוספה למנוע של SQL Server לתמיכה באופרטור APPLY. כעת מותר לך להעביר פרמטר המתייחס לטור משאילתה חיצונית לפונקציה טבלאית. כדוגמה ליכולת זו, השאילתה הבאה מחזירה לכל ספק את המחיר הנמוך מבין שני המשקאות היקרים ביותר (בהנחה שיש לפחות שניים), ומייצרת את הקלט המוצג בטבלה 7–7:

```
SELECT S.SupplierID, CompanyName,
  (SELECT MIN(UnitPrice)
   FROM dbo.fn_top_products(S.SupplierID, 1, 2) AS P) AS Price
FROM dbo.Suppliers AS S;
```

טבלה 7–7: המחיר הנמוך מבין שני המשקאות היקרים ביותר לכל ספק

SupplierID	*CompanyName*	*Price*
18	Aux joyeux ecclésiastiques	18.00
16	Bigfoot Breweries	14.00
5	Cooperativa de Quesos 'Las Cabras'	NULL
27	Escargots Nouveaux	NULL
1	Exotic Liquids	18.00
29	Forêts d'érables	NULL
14	Formaggi Fortini s.r.l.	NULL
28	Gai pâturage	NULL
24	G'day, Mate	NULL
3	Grandma Kelly's Homestead	NULL
11	Heli Süßwaren GmbH & Co. KG	NULL
23	Karkki Oy	18.00
20	Leka Trading	46.00
21	Lyngbysild	NULL
25	Ma Maison	NULL
6	Mayumi's	NULL
19	New England Seafood Cannery	NULL
2	New Orleans Cajun Delights	NULL
13	Nord-Ost-Fisch Handelsgesellschaft mbH	NULL
15	Norske Meierier	NULL

SupplierID	*CompanyName*	*Price*
26	Pasta Buttini s.r.l.	NULL
7	Pavlova, Ltd.	15.00
9	PB Knäckebröd AB	NULL
12	Plutzer Lebensmittelgroßmärkte AG	7.75
10	Refrescos Americanas LTDA	4.50
8	Specialty Biscuits, Ltd.	NULL
17	Svensk Sjöföda AB	NULL
4	Tokyo Traders	NULL
22	Zaanse Snoepfabriek	NULL

פתרונות לבעיות נפוצות המשתמשים ב–TOP ו–APPLY

כעת לאחר שכיסינו את העקרונות של TOP ו–APPLY, אציג בעיות נפוצות ופתרונות המשתמשים ב–TOP ו–APPLY.

TOP n לכל קבוצה

בפרק 4 ובפרק 6 דנתי בבעיה המערבת שוברי–שוויון, כאשר התבקשת להחזיר את ההזמנה האחרונה לכל עובד. בעיה זו היא למעשה מקרה מיוחד של בעיה כללית יותר, בה אתה מעוניין ב–n השורות הראשונות בכל קבוצה – למשל, החזרת שלוש ההזמנות האחרונות לכל עובד. שוב, להזמנות בעלות ערכי OrderDate גבוהים יותר יש קדימות, אך עליך להציג שובר–שוויון כדי לקבוע קדימות במקרה של מספר הזמנות בעלות אותו ערך OrderDate. כאן אשתמש ב–OrderID מקסימלי כשובר שוויון. אציג פתרונות לסוג בעיות זה המשתמשים ב–TOP ו–APPLY. תמצא שפתרונות אלו פשוטים משמעותית מאלו שהצגתי קודם, ובמקרים מסוימים הם משמעותית מהירים יותר. עם זאת, כללי אינדקסים נשארים זהים. כלומר, תרצה אינדקס כאשר רשימת המפתחות היא טורי החציצה (EmployeeID), טורי המיון (OrderDate), טורי שובר–השוויון (OrderID), ולמטרות כיסוי הטורים האחרים המוזכרים בשאילתה כרשימת הטורים הכלולים (RequiredDate ו–CustomerID). כמובן, ב– SQL Server 2000, שאינו תומך בטורים כלולים שאינם מפתח (פסוקית INCLUDE), עליך להוסיף את אלו לרשימת המפתחות.

בטרם נעבור על הפתרונות השונים, הרץ את הקוד הבא כדי לייצר את האינדקסים הרצויים על הטבלאות Orders ו– [Order Details] המשתתפות בדוגמאות שלי:

```
CREATE UNIQUE INDEX idx_eid_od_oid_i_cid_rd
  ON dbo.Orders(EmployeeID, OrderDate, OrderID)
     INCLUDE(CustomerID, RequiredDate);

CREATE UNIQUE INDEX idx_oid_qtyd_pid
  ON dbo.[Order Details](OrderID, Quantity DESC, ProductID);
```

הפתרון הראשון שאציג ימצא את ההזמנה האחרונה לכל עובד. הפתרון מבצע שאילתה על טבלת Orders, ומסנן רק הזמנות בעלות ערך OrderID השווה לתוצאת תת–השאילתה. תת–השאילתה מחזירה את ערך ה–OrderID של ההזמנה האחרונה ביותר עבור העובד הנוכחי על ידי שימוש בלוגיקה פשוטה של TOP(1). קטע–קוד 1–7 מציג את שאילתת הפתרון, ומייצר את הפלט המוצג בטבלה 8–7 ואת תוכנית העבודה המוצגת בתרשים 1–7.

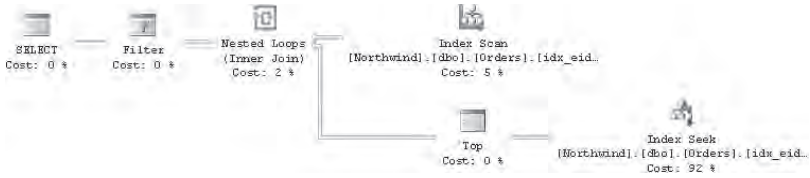
קטע–קוד 1–7: פתרון 1 לבעיית ההזמנה האחרונה לכל עובד

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderID =
  (SELECT TOP(1) OrderID
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID
   ORDER BY OrderDate DESC, OrderID DESC);
```

טבלה 8–7: הזמנה אחרונה לכל עובד

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*	*RequiredDate*
11077	RATTC	1	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11073	PERIC	2	1998-05-05 00:00:00.000	1998-06-02 00:00:00.000
11063	HUNGO	3	1998-04-30 00:00:00.000	1998-05-28 00:00:00.000
11076	BONAP	4	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11043	SPECD	5	1998-04-22 00:00:00.000	1998-05-20 00:00:00.000
11045	BOTTM	6	1998-04-23 00:00:00.000	1998-05-21 00:00:00.000
11074	SIMOB	7	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11075	RICSU	8	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11058	BLAUS	9	1998-04-29 00:00:00.000	1998-05-27 00:00:00.000

תרשים 7–1: תוכנית עבודה עבור השאילתה בקטע–קוד 7–1



לפתרון זה יש מספר יתרונות על פני הפתרון שהצגתי קודם בספר. בהשוואה לפתרון תת–השאילתות של ANSI שהצגתי בפרק 4, פתרון זה מהיר יותר ופשוט הרבה יותר, במיוחד כאשר יש לך מספר טורי מיון ושובר–שוויון; אתה פשוט מרחיב את רשימת ה– ORDER BY בתת–השאילתה כך שתכלול את הטורים הנוספים. בהשוואה לפתרון המבוסס על צבירות שהצגתי בפרק 6, פתרון זה איטי יותר אך פשוט הרבה יותר.

בחן את תוכנית העבודה של השאילתה, המוצגת בתרשים 7–1. האופרטור Index Scan מראה שהאינדקס–המכסה idx_eid_od_oid_i_cid_rd נסרק פעם אחת. הענף התחתון של האופרטור Nested Loops מייצג את העבודה שנעשתה עבור כל שורה שהוחזרה מה– Index Scan. כאן אתה רואה שעבור כל שורה מה– Index Scan, מתרחשות פעולות Index Seek ו–TOP כדי למצוא את ההזמנה האחרונה של העובד הנתון. זכור שרמת העלה של האינדקס מחזיקה את הנתונים ממוינים לפי EmployeeID, OrderDate, OrderID, בסדר זה (משמאל לימין); המשמעות היא שהשורה האחרונה בכל קבוצת שורות לכל עובד מייצגת את העובד הנוכחי, והאופרטור TOP הולך צעד אחורה כדי להחזיר את המפתח של ההזמנה האחרונה. אז אופרטור סינון שומר רק הזמנות בהן ערך ה–OrderID החיצוני מתאים לזה המוחזר על ידי תת–השאילתה.

עלות ה–I/O של שאילתה זו היא 1,787 קריאות לוגיות, ומספר זה מתחלק כלהלן: הסריקה המלאה של האינדקס–המכסה דורשת 7 קריאות לוגיות, מכיוון שהאינדקס מתפרש על פני 7 דפי נתונים; כל אחד מ–830 ה– index seeks דורש לפחות 2 קריאות לוגיות, מכיוון שהאינדקס הוא בעל 2 רמות, וכמה מה– index seeks דורשים 3 קריאות לוגיות בסך הכול, מכיוון שה–seek עשוי להוביל להתחלה של דף נתונים אחד וה–OrderID המאוחר ביותר עשוי להיות בסוף הדף הקודם.

אם הבנת שפעולת seek נפרדת בתוך האינדקס נקראה עבור כל הזמנה חיצונית, תוכל לנחש שיש כאן מקום לאופטימיזציה. פוטנציאל הביצועים הוא לקרוא ל–seek יחיד בלבד לכל עובד, ולא לכל הזמנה, מכיוון שבסופו של דבר אתה מעוניין בהזמנה האחרונה לכל עובד. אתאר כיצד להשיג אופטימיזציה כזו מייד. אך לפני כן, ארצה להצביע על יתרון נוסף של פתרון זה על פני האחרים שהצגתי קודם בספר. הפתרונות הקודמים היו מוגבלים להחזרת הזמנה אחת בלבד לכל עובד, ואילו פתרון זה ניתן להרחבה בקלות, לתמיכה בכל מספר הזמנות לכל עובד, על ידי המרת אופרטור השוויון בביטוי IN. שאילתת הפתרון מוצגת בקטע–קוד 7–2.

קטע-קוד 7-2: פתרון 1 לבעיית הזמנות אחרונות לכל עובד

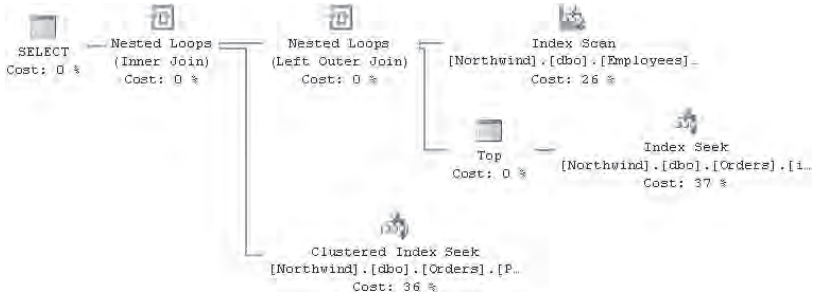
```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderID IN
  (SELECT TOP(3) OrderID
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID
   ORDER BY OrderDate DESC, OrderID DESC);
```

כעת, הבה נעבור לאופטימיזציית הפתרון. זכור שאתה מנסה לתת ל-optimizer רמז שאתה מעוניין בפעולת index seek אחת לכל עובד, ולא אחת לכל הזמנה. תוכל להשיג זאת על ידי ביצוע שאילתה על טבלת Employees והשגת ה-OrderID המאוחר ביותר לכל עובד. צור טבלה נגזרת משאילתה זו מול Employees, ובצע join לטבלה הנגזרת עם טבלת Orders לפי ערכי OrderID תואמים. קטע-קוד 7-3 מציג את שאילתת הפתרון, ומפיק את תוכנית העבודה המוצגת בתרשים 7-2.

קטע-קוד 7-3: פתרון 2 לבעיית ההזמנה האחרונה לכל עובד

```
SELECT O.OrderID, CustomerID, O.EmployeeID, OrderDate, RequiredDate
FROM (SELECT EmployeeID,
        (SELECT TOP(1) OrderID
         FROM dbo.Orders AS O2
         WHERE O2.EmployeeID = E.EmployeeID
         ORDER BY OrderDate DESC, OrderID DESC) AS TopOrder
      FROM dbo.Employees AS E) AS EO
  JOIN dbo.Orders AS O
    ON O.OrderID = EO.TopOrder;
```

תרשים 7-2: תוכנית עבודה עבור השאילתה בקטע-קוד 7-3



תוכל לראות בתוכנית שאחד מהאינדקסים על טבלת Employees נסרק כדי לגשת לקודי העובדים. האופרטור הבא המופיע בתוכנית (Nested Loops) מפעיל seek באינדקס על Orders כדי לקבל את קודי ההזמנות האחרונות של העובדים. עם 9 עובדים, יבוצעו רק 9 פעולות seek, בהשוואה ל–830 מקודם שנבעו ממספר ההזמנות. לבסוף, אופרטור Nested Loops נוסף, מפעיל seek אחד לכל עובד באינדקס–clustered על Orders.OrderID כדי לחפש את מאפייני ההזמנה מה–OrderID. אם האינדקס על OrderID לא היה clustered, היית רואה lookup נוסף הניגש לשורת הנתונים המלאה. עלות ה–I/O של שאילתה זו היא 36 קריאות לוגיות בלבד.

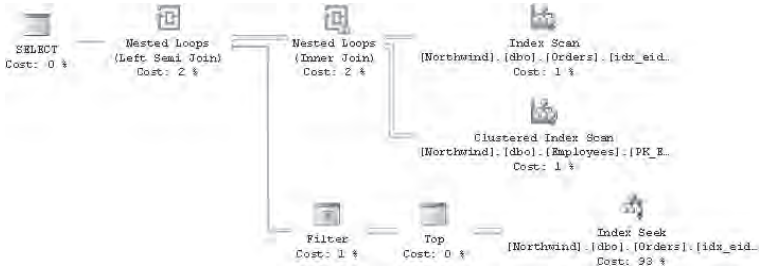
ניסיון לשחזר את אותה הצלחה כאשר אתה זקוק ליותר מהזמנה אחת לכל עובד – מאכזב. מכיוון שאינך יכול להחזיר יותר ממפתח אחד ברשימת ה–SELECT על ידי שימוש בתת–שאילתה, אתה עשוי לנסות לעשות דבר דומה בתנאי join בין Employees ו–Orders. שאילתת הפתרון מוצגת בקטע–קוד 4–7.

קטע–קוד 4–7: פתרון 3 לבעיית 3 ההזמנות האחרונות לכל עובד

```
SELECT OrderID, CustomerID, E.EmployeeID, OrderDate, RequiredDate
FROM dbo.Employees AS E
   JOIN dbo.Orders AS O1
     ON OrderID IN
       (SELECT TOP(3) OrderID
        FROM dbo.Orders AS O2
        WHERE O2.EmployeeID = E.EmployeeID
        ORDER BY OrderDate DESC, OrderID DESC);
```

אך פתרון זה מניב את התוכנית הבלתי–מוצלחת המוצגת בתרשים 3–7, המייצרת 15,897 קריאות לוגיות מול טבלת Orders ו–1661 קריאות לוגיות מול טבלת Employees. במקרה זה, עדיף לך להשתמש בפתרון שהצגתי קודם התומך בהחזרת מספר הזמנות לכל עובד.

תרשים 3–7: תוכנית עבודה לשאילתה בקטע–קוד 4–7

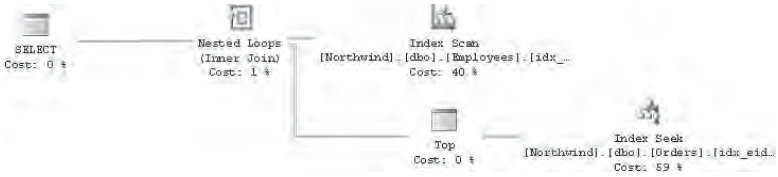


עד כה, כל הפתרונות היו תואמי SQL Server 2000 (מלבד העובדה שב– SQL Server 2000 אינך משתמש בסוגריים ב–TOP, כמובן). ב– SQL Server 2005, באפשרותך להשתמש באופרטור APLLY בפתרון שביצועיו טובים יותר מכל הפתרונות האחרים שהצגתי עד כה, ואשר תומך גם בהחזרת מספר שורות לכל עובד. אתה מפעיל על טבלת Employees ביטוי טבלה המחזיר, לשורה נתונה בטבלת Employees, את n ההזמנות האחרונות לעובד בשורה זו. קטע–קוד 5–7 מציג את שאילתת הפתרון, ומפיק את תוכנית העבודה המוצגת בתרשים 4–7.

קטע–קוד 5–7: פתרון 4 לבעיית הזמנות אחרונות לכל עובד

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Employees AS E
  CROSS APPLY
    (SELECT TOP(3) OrderID, CustomerID, OrderDate, RequiredDate
     FROM dbo.Orders AS O
     WHERE O.EmployeeID = E.EmployeeID
     ORDER BY OrderDate DESC, OrderID DESC) AS A;
```

תרשים 4–7: תוכנית עבודה עבור השאילתה המוצגת בקטע–קוד 5–7



התוכנית סורקת אינדקס על טבלת Employees עבור ערכי EmployeeID. כל ערך EmployeeID מפעיל seek יחיד בתוך האינדקס–המכסה על Orders כדי להחזיר את 3 ההזמנות האחרונות המבוקשות לכל עובד. החלק המעניין כאן הוא שאינך מקבל רק את המפתחות של השורות שנמצאו; אלא, תוכנית זו מאפשרת החזרת מספר טורים. כך שאין צורך בפעילויות נוספות כדי להחזיר את הטורים שאינם מפתח. עלות ה–I/O של שאילתה זו היא 18 קריאות לוגיות בלבד.

באופן מפתיע, קיים פתרון שיכול להיות אף מהיר יותר מזה המשתמש באופרטור APPLY בנסיבות מסוימות, אותן אתאר מייד. הפתרון משתמש בפונקציה ROW_NUMBER. אתה מחשב את מספר השורה של כל הזמנה, עם חציצה לפי EmployeeID, ובהתבסס על סדר OrderDate DESC, OrderID DESC. אז, בשאילתה חיצונית, אתה מסנן רק תוצאות עם מספר שורה נמוך או שווה ל–3. האינדקס האופטימלי לפתרון זה דומה לאינדקס–המכסה שהוצג קודם, אך הטורים OrderDate ו–OrderID מצוינים בו בסדר יורד:

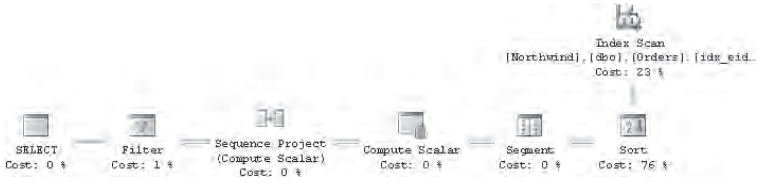
```
CREATE UNIQUE INDEX idx_eid_odD_oidD_i_cid_rd
  ON dbo.Orders(EmployeeID, OrderDate DESC, OrderID DESC)
     INCLUDE(CustomerID, RequiredDate);
```

קטע–קוד 7–6 מציג את שאילתת הפתרון, ומפיק את תוכנית העבודה המוצגת בתרשים 7–5.

קטע–קוד 7–6: פתרון 5 לבעיית ההזמנות האחרונות לכל עובד

```
SELECT OrderID, CustomerID, OrderDate, RequiredDate
FROM (SELECT OrderID, CustomerID, OrderDate, RequiredDate,
        ROW_NUMBER() OVER(PARTITION BY EmployeeID
                          ORDER BY OrderDate DESC, OrderID DESC) AS RowNum
      FROM dbo.Orders) AS D
WHERE RowNum <= 3;
```

תרשים 7–5: תוכנית עבודה עבור השאילתה בקטע–קוד 7–6



תיארתי כבר את תוכנית העבודה המופקת עבור פונקציות דירוג בפרק 4, ותוכנית זו דומה מאוד. עלות ה–I/O כאן היא 7 קריאות לוגיות בלבד הנובעות מהסריקה המלאה היחידה של האינדקס–המכסה. שים לב שכדי לחשב את מספרי השורה כאן, על האינדקס להיסרק בצורה מלאה. כאשר אתה מחפש אחוז קטן של שורות בכל קבוצה, בטבלאות גדולות, האופרטור APPLY יהיה מהיר יותר מכיוון שהעלות הכוללת של פעולות ה–seek המרובות, אחת לכל קבוצה, תהיה נמוכה מאשר הסריקה המלאה של האינדקס–המכסה.

קיים יתרון חשוב לפתרונות המשתמשים באופרטור APPLY ובפונקציה ROW_NUMBER על פני פתרונות תואמי SQL Server 2000 המשתמשים ב–TOP. פתרונות תואמי SQL Server 2000 נתמכים רק כאשר לטבלה הנתונה יש טור מפתח יחיד, מכיוון שהם נשענים על תת–שאילתה המחזירה סקלאר (scalar). הפתרונות החדשים, לעומת זאת, ישימים באותה מידה עם מפתחות מורכבים. למשל, נאמר שאתה מעוניין בשלושת הפריטים הראשונים בכל הזמנה, כאשר הקדימות נקבעת על ידי Quantity DESC, וכאשר ProductID ASC משמש כסדר של שובר–השוויון. לטבלת [Order Details] יש מפתח ראשי מורכב, (OrderID,ProductID), כך שאינך יכול להחזיר מפתח לטבלה זו מתת–שאילתה. האופרטור APPLY עם זאת, אינו נשען על קיומו של מפתח בעל טור אחד. הוא מעוניין

רק בקשר בין הטבלה [Order Details] הפנימית לבין טבלת Orders החיצונית בהתבסס על התאמה של OrderID ועל מיון המבוסס על Quantity DESC ו– ProductID ASC:

```
SELECT D.OrderID, ProductID, Quantity
FROM dbo.Orders AS O
  CROSS APPLY
    (SELECT TOP(3) OD.OrderID, ProductID, Quantity
     FROM [Order Details] AS OD
     WHERE OD.OrderID = O.OrderID
     ORDER BY Quantity DESC, ProductID) AS D;
```

באופן דומה, הפתרון המתבסס על ROW_NUMBER אינו נשען על קיומו של מפתח בעל טור יחיד. הוא פשוט מחשב מספרי שורה המוקצים במחיצות לפי OrderID, וממוינים לפי Quantity DESC ו– ProductID ASC:

```
SELECT OrderID, ProductID, Quantity
FROM (SELECT ROW_NUMBER() OVER(PARTITION BY OrderID
                               ORDER BY Quantity DESC, ProductID) AS RowNum,
        OrderID, ProductID, Quantity
      FROM dbo.[Order Details]) AS D
WHERE RowNum <= 3;
```

התאמת מופעים נוכחיים וקודמים

התאמת מופעים נוכחיים וקודמים היא בעיה נוספת שלפתרונה אתה יכול להשתמש ב–TOP. הבעיה היא התאמה לכל שורה "נוכחית", שורה מאותה טבלה הנחשבת לשורה "הקודמת", בהתבסס על קריטריון מיון מסוים – לרוב, מבוסס זמן. בקשה זו משרתת את הצורך לערוך חישובים המערבים מדידות הן משורה "נוכחית" והן משורה "קודמת". דוגמאות לבקשות כאלה הן חישובי מגמות, הפרשים, אחוזים, וכך הלאה. כאשר עליך לכלול ערך אחד בלבד מהשורה הקודמת בחישוב שלך, השתמש בשאילתת TOP(1) פשוטה להשגת ערך זה. אך כאשר אתה צריך מספר מדידות מהשורה הקודמת, הגיוני יותר, במושגים של ביצועים, להשתמש ב–join, מאשר במספר תת–שאילתות.

נניח שעליך להתאים כל הזמנה של עובד עם ההזמנה הקודמת שלו, על ידי שימוש ב–OrderDate כדי לקבוע מה ההזמנה הקודמת וב–OrderID כשובר–השוויון. ברגע שנמצאה התאמה להזמנות של העובד, תוכל לבקש חישובים המערבים טורים משני הצדדים – למשל, חישוב הפרשים בתאריך בין ההזמנה הנוכחית והקודמת, תאריכים נדרשים, וכך הלאה. כדי לקצר, לא אראה את החישובים בפועל של ההפרשים; אלא אתמקד רק בשיטות ההתאמה. פתרון אחד הוא לבצע join בין שני מופעים של טבלת Orders: אחד המייצג את השורות הנוכחיות (Cur), והשני מייצג את השורות הקודמות (Prv). תנאי ה–join יתאים את Prv.OrderID ל–OrderID של ההזמנה הקודמת, אותה אתה מחזיר

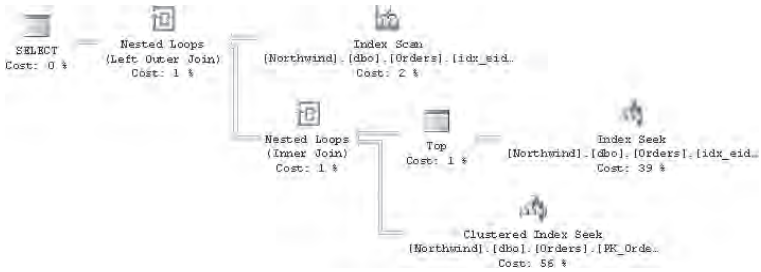
משאילתת TOP(1). אתה משתמש ב– LEFT OUTER join כדי לשמור את ההזמנה ה"ראשונה" לכל עובד. inner join יעלים הזמנות כאלו מכיוון שלא תמצא עבורן התאמה. קטע–קוד 7–7 מציג את שאילתת הפתרון לבעיית ההתאמה.

קטע–קוד 7–7: שאילתת פתרון 1 לבעיית התאמת מופעים נוכחי וקודם

```
SELECT Cur.EmployeeID,
  Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
  Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
  Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM dbo.Orders AS Cur
  LEFT OUTER JOIN dbo.Orders AS Prv
    ON Prv.OrderID =
       (SELECT TOP(1) OrderID
        FROM dbo.Orders AS O
        WHERE O.EmployeeID = Cur.EmployeeID
          AND (O.OrderDate < Cur.OrderDate
               OR (O.OrderDate = Cur.OrderDate
                   AND O.OrderID < Cur.OrderID))
        ORDER BY OrderDate DESC, OrderID DESC)
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

המסנן של תת–השאילתה ערמומי מעט, מכיוון שהסדר/קדימות נקבעים על ידי שני טורים: OrderDate (טור מיון) ו–OrderID (שובר–שוויון). אם הבקשה הייתה לקדימות המבוססת על טור אחד – נאמר, OrderID בלבד – המסנן היה פשוט הרבה יותר – O.OrderID < Cur.OrderID. מכיוון שמעורבים שני טורים, שורות "קודמות" מזוהות על ידי ביטוי לוגי האומר:
inner_sort_col < outer_sort_col or
(inner_sort_col = outer_sort_col and inner_tiebreaker < outer_tiebreaker).
שאילתה זו מייצרת את תוכנית העבודה המוצגת בתרשים 6–7, עם עלות I/O של 3,533 קריאות לוגיות.

תרשים 6–7: תוכנית עבודה לשאילתה בקטע–קוד 7–7



תחילה התוכנית סורקת את האינדקס–המכסה שיצרתי קודם על רשימת המפתח (EmployeeID, OrderDate, OrderID), כאשר הטורים המכוסים (CustomerID, RequiredDate) מצוינים כטורים כלולים. המטרה של סריקה זו היא להחזיר את השורות "הנוכחיות". לכל שורה נוכחית, אופרטור Nested Loops מאתחל פעולת Index Seek באותו אינדקס. פעולת ה–Index Seek מופעלת על ידי תת–השאילתה ומטרתה להשיג את המפתח (OrderID) של השורה "הקודמת". לכל OrderID קודם שמוחזר, אופרטור Nested Loops נוסף מביא את רשימת הטורים הרצויה מהשורה הקודמת. אתה מבין בוודאי שאחת משתי פעולות ה–seek מיותרת ושקיים פוטנציאל לשאילתה אופטימלית יותר אשר תניב seek אחד בלבד לכל הזמנה.

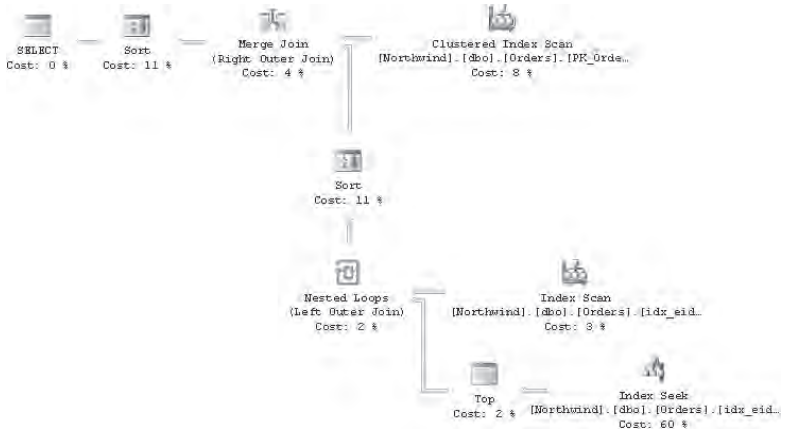
ב– SQL Server 2000, תוכל לנסות מספר שינויים לשאילתה שעשויים לשפר ביצועים. אך מכיוון שבקשה עם האפשרות TOP לכל שורה חיצונית, יכולה להיות מאותחלת רק בתת–שאילתה ולא בביטוי טבלה, לא פשוט להימנע משתי פעילויות נפרדות לכל הזמנה כאשר משתמשים ב–TOP. קטע–קוד 8–7 מציג דוגמה של שינוי שאילתה המאפשרת את התוכנית לאחר אופטימיזציה המוצגת בתרשים 7–7, כאשר השאילתה רצה ב– SQL Server 2005.

קטע–קוד 8–7: שאילתת פתרון 2 לבעיית התאמת מופעים נוכחי וקודם

```
SELECT Cur.EmployeeID,
  Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
  Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
  Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM (SELECT EmployeeID, OrderID, OrderDate, RequiredDate,
        (SELECT TOP(1) OrderID
         FROM dbo.Orders AS O2
         WHERE O2.EmployeeID = O1.EmployeeID
           AND (O2.OrderDate < O1.OrderDate
                OR O2.OrderDate = O1.OrderDate
                   AND O2.OrderID < O1.OrderID)
         ORDER BY OrderDate DESC, OrderID DESC) AS PrvOrderID
      FROM dbo.Orders AS O1) AS Cur
  LEFT OUTER JOIN dbo.Orders AS Prv
    ON Cur.PrvOrderID = Prv.OrderID
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

לתוכנית זו עלות I/O של 2,033 קריאות לוגיות (למרות שב– SQL Server 2000 תקבל תוכנית אחרת עם עלות I/O גבוהה יותר). הפתרון יוצר טבלה נגזרת הנקראת Cur המכילה הזמנות נוכחיות, עם טור נוסף (PrvOrderID) המחזיק את ה–OrderID של ההזמנה הקודמת, כפי שהושג על ידי תת–שאילתה קשורה. השאילתה החיצונית אז מבצעת join בין Cur לבין מופע נוסף של Orders, המקבל את הכינוי Prv ומתייחס לרשימת הטורים המלאה מההזמנה הקודמת. שיפור הביצועים ועלות ה–I/O הנמוכה הם בעיקר התוצאה של האלגוריתם merge join בו משתמשת התוכנית.

תרשים 7–7: תוכנית עבודה לשאילתה בקטע–קוד 8–7



בתוכנית השאילתה הגרפית, הקלט העליון לאופרטור Merge Join הוא תוצאה של סריקה ממוינת של האינדקס–clustered על OrderID, המייצג את ההזמנות "הקודמות", וזהו הצד הלא–שמור של ה– outer join. הקלט הנמוך הוא התוצאה של סריקת האינדקס–המכסה והבאת כל OrderID קודם עם פעולת seek שלאחריה Top 1. כדי להכין קלט זה למיזוג, התוכנית ממיינת את השורות לפי OrderID.

merge join כדאי כאן מבחינת העלות מכיוון שהשורות בטבלת Orders עברו מיון מקדים על טור המפתח של האינדקס–clustered ולא הייתה זו יותר מדי עבודה למיין את הקלט השני בהכנה למיזוג. במערכות תפעוליות גדולות יותר, סביר להניח שפני הדברים יהיו שונים. עם מספר גדול בהרבה של שורות ואינדקס–clustered שונה – על טור שלעיתים קרובות מופיע בשאילתות תחום – אל לך לצפות לתוכנית שאילתה זהה.

שים לב: זכור שערכם של דיוני הביצועים שאני עורך בספר, הוא בהבנה כיצד לקרוא תוכניות וכיצד לכתוב שאילתות ביותר מאשר דרך אחת. ייתכן שתקבל תוכניות עבודה שונות בגרסאות שונות של SQL Server – כמו במקרה של השאילתה בקטע–קוד 8–7. הפסקה האחרונה תיארה את התוכנית שתקבל ב– SQL Server 2005, השונה מזו שתקבל ב– SQL Server 2000. בדומה, תוכניות עבודה יכולות להשתנות בין רמות service pack שונות של המוצר. כמו כן, זכור שתוכניות עבודה עשויות להשתנות כאשר פיזור הנתונים משתנה.

זהו המקום בו כדאי להשתמש באופרטור APPLY. שימוש כזה מוביל לעיתים קרובות לתוכניות פשוטות ויעילות בעלות ביצועים טובים אפילו בנפחי נתונים גדולים. שימוש באופרטור APPLY במקרה זה מוביל לתוכנית הסורקת את הנתונים פעם אחת כדי להגיע להזמנות הנוכחיות, ומבצעת index seek יחיד לכל הזמנה נוכחית כדי להביא מהאינדקס–המכסה את כל טורי ההזמנה הקודמת בבת–אחת.

קטע–קוד 7–9: שאילתת פתרון 3 לבעיית התאמת מופעים נוכחי וקודם

```
SELECT Cur.EmployeeID,
  Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
  Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
  Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM dbo.Orders AS Cur
  OUTER APPLY
    (SELECT TOP(1) OrderID, OrderDate, RequiredDate
     FROM dbo.Orders AS O
     WHERE O.EmployeeID = Cur.EmployeeID
       AND (O.OrderDate < Cur.OrderDate
            OR (O.OrderDate = Cur.OrderDate
               AND O.OrderID < Cur.OrderID))
     ORDER BY OrderDate DESC, OrderID DESC) AS Prv
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

תרשים 7–8: תוכנית עבודה לשאילתה בקטע–קוד 7–9

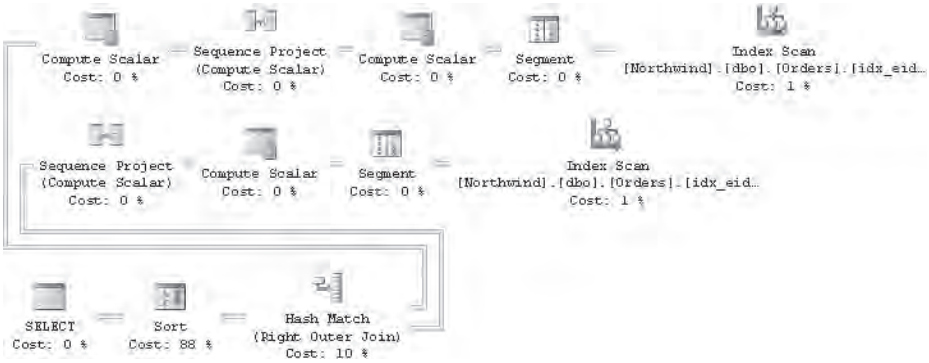
אך אם אינך מסתפק בביצועים "סבירים" ומעוניין בפתרון מעולה באמת, תזדקק לפונקציה ROW_NUMBER. אתה יוצר CTE המחשב מספרי שורה עבור הזמנות עם מחיצות לפי EmployeeID ובהתבסס על סדר OrderDate, OrderID. אתה מבצע join על שני מופעים של ה–CTE, אחד מייצג את ההזמנות הנוכחיות והשני מייצג את ההזמנות הקודמות. תנאי ה–join יתבסס על התאמה של ערכי EmployeeID ומספרי שורה שביניהם הפרש של אחד. קטע–קוד 7–10 מציג את שאילתת הפתרון, ומפיק את תוכנית העבודה המוצגת בתרשים 7–9.

קטע–קוד 7–10: שאילתת פתרון 4 לבעיית התאמת מופעים נוכחי וקודם

```
WITH OrdersRN AS
(
  SELECT EmployeeID, OrderID, OrderDate, RequiredDate,
    ROW_NUMBER() OVER(PARTITION BY EmployeeID
                      ORDER BY OrderDate, OrderID) AS rn
  FROM dbo.Orders
)
```

```
SELECT Cur.EmployeeID,
  Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
  Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
  Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM OrdersRN AS Cur
  LEFT OUTER JOIN OrdersRN AS Prv
    ON Cur.EmployeeID = Prv.EmployeeID
    AND Cur.rn = Prv.rn + 1
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

תרשים 9–7: תוכנית עבודה לשאילתה בקטע–קוד 10–7



מכיוון שהתוכנית סורקת את האינדקס–המכסה פעמיים בלבד כדי לגשת לטורי ההזמנה ולחשב את מספרי השורה, עלות ה–I/O הכוללת שלה היא 14 קריאות לוגיות, והיא משאירה את כל הפתרונות האחרים הרחק מאחור מבחינת ביצועים.

כדי לנקות, הרץ את הקוד הבא, המסיר את האינדקסים ששימשו לפתרונות שהוצגו כאן:

```
DROP INDEX dbo.Orders.idx_eid_od_oid_i_cid_rd;
DROP INDEX dbo.Orders.idx_eid_odD_oidD_i_cid_rd;
DROP INDEX dbo.[Order Details].idx_oid_qtyd_pid;
```

דפדוף (Paging)

התחלתי לדון בדפדוף בפרק 4, כשהצגתי פתרונות המבוססים על מספרי שורה. כתזכורת, אתה מעוניין להחזיר שורות מסט התוצאה של שאילתה בדפים או מקטעים, כשאתה מאפשר למשתמש לנווט בין הדפים. בדוגמאות שלי, השתמשתי בטבלת Orders במסד הנתונים Northwind.

בסביבות תפעוליות, דפדוף מערב לרוב מסננים דינמיים ומיון המתבסס על בקשות המשתמש. כדי להתמקד בשיטות הדפדוף, אניח שאין כאן מסננים ושהסדר המבוקש הוא

לפי OrderDate עם OrderID כשובר–שוויון. האינדקס האופטימלי לפתרונות הדפדוף שאציג, ממלא אחר קווים מנחים דומים לפתרונות TOP אחרים שהצגתי – כלומר, אינדקס על טור או טורי המיון ועל טור או טורי שובר–השוויון. אם אתה יכול לעמוד במחיר, צור את האינדקס כאינדקס–מכסה, או על ידי יצירתו כאינדקס–clustered של הטבלה, או אם הוא אינו clustered, על ידי הכללת הטורים האחרים המוזכרים בשאילתה. זכור מפרק 3 שב– SQL Server 2005 אינדקס יכול להכיל טורים שאינם–מפתח, המצוינים בפסוקית INCLUDE של הפקודה CREATE INDEX. הטורים שאינם–מפתח של אינדקס מופיעים רק ברמת העלה של האינדקס. אתה חייב להוסיף טורים נוספים לרשימת המפתח אם ברצונך שהאינדקס יכסה את השאילתה, או שאתה חייב ליצור את האינדקס כאינדקס–clustered של הטבלה. אם אינך יכול לעמוד במחיר של אינדקס–מכסה, ודא לפחות שאתה יוצר אינדקס על טורי ה– sort+tiebreaker. התוכניות תהיינה יעילות פחות מאשר עם אינדקס–מכסה מכיוון שיהיו מעורבים lookups כדי לקבל את שורת הנתונים, אך לפחות לא תקבל סריקת טבלה לכל בקשת דף.

כדי למיין לפי OrderDate ו–OrderID, וכדי לכסות את הטורים CustomerID ו–EmployeeID, צור את האינדקס הבא:

```
CREATE INDEX idx_od_oid_i_cid_eid
  ON dbo.Orders(OrderDate, OrderID) INCLUDE(CustomerID, EmployeeID);
```

שים לב: ב– SQL Server 2000, זכור לכלול את כל הטורים כחלק מרשימת המפתח, שכן הפסוקית INCLUDE נוספה רק ב– SQL Server 2005.

הפתרון שאציג כאן תומך בדפדוף בין דפים רציפים. כלומר, אתה מבקש את הדף הראשון ואז ממשיך לבא. ייתכן שתרצה לספק גם את האפשרות לבקש דף קודם. מומלץ בחום ליישם את הבקשות לדף הראשון, הדף הבא והדף הקודם כפרוצדורות מאוחסנות, הן מסיבות של ביצועים והן מסיבות של עיטוף. בצורה כזו אתה יכול לקבל שימוש–מחדש יעיל של תוכנית, ותמיד באפשרותך לשנות את היישום של הפרוצדורה המאוחסנת אם אתה מוצא שיטות יעילות יותר, ללא השפעה על משתמשי הפרוצדורה המאוחסנת.

דף ראשון

יישום הפרוצדורה המאוחסנת המחזירה את הדף הראשון הוא באמת פשוט, מכיוון שאינך צריך עוגן כדי לסמן את נקודת ההתחלה שלך. אתה פשוט מחזיר את מספר השורות המבוקשות מלמעלה, כלהלן:

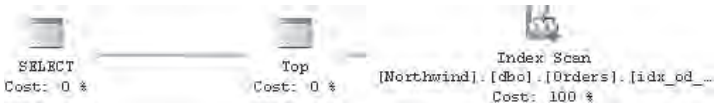
```
CREATE PROC dbo.usp_firstpage
    @n AS INT = 10
AS
SELECT TOP(@n) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate, OrderID;
GO
```

שים לב: בדוגמה זו, ל– ORDER BY שתי מטרות: להגדיר מה המשמעות של TOP, ולשלוט בסדר השורות בסט התוצאה.

קיומו של אינדקס על טורי המיון, במיוחד אם הוא אינדקס–מכסה כמו שיצרתי למטרה זו, מאפשר תוכנית אופטימלית שבה רק דפי השורות הרלוונטיים נסרקים בתוך האינדקס לפי הסדר. תוכל לראות זאת על ידי הרצה של הפרוצדורה המאוחסנת הבאה ובחינת התוכנית המופיעה בתרשים 10–7:

```
EXEC dbo.usp_firstpage;
```

תרשים 10–7: תוכנית עבודה עבור פרוצדורה מאוחסנת usp_firstpage



שורות נסרקות בתוך האינדקס, מראש הרשימה המקושרת והלאה לפי הסדר. האופרטור TOP מפסיק את הסריקה ברגע שהושג מספר השורות המבוקש.

הדף הבא

הבקשה לדף "הבא" חייבת להישען על שורת עוגן מסוימת המסמנת היכן הדף צריך להתחיל. עוגן זה צריך להיות מסופק לפרוצדורה המאוחסנת כקלט. העוגן יכול להיות ערכי טור המיון של השורה האחרונה בדף הקודם, מכיוון שכפי שתוכל לזכור, למטרות דטרמיניזם ערכי המיון חייבים להיות ייחודיים. ביישום הלקוח, קיבלת כבר את הדף הקודם. כאשר אתה מקבל בקשה לדף הבא, תוכל לספק את אלו כקלט לפרוצדורה המאוחסנת.

כיוון שאתה זוכר כי בפועל מסננים ומיון הם לרוב דינמיים, אינך יכול להישען על אף מספר מסוים או סוג טורים כפרמטרי קלט. כך שעיצוב חכם יותר, אשר יאפשר שיפורים עתידיים של הפרוצדורה לתמיכה בהפעלה דינמית, יהיה לספק את המפתח הראשי של שורת העוגן כקלט, ולא את ערכי טורי המיון. יישום הלקוח ישמור בצד את ערך המפתח הראשי מהשורה האחרונה שקיבל וישתמש בו כקלט להפעלה הבאה של הפרוצדורה המאוחסנת.

להלן היישום של הפרוצדורה usp_nextpage:

```
CREATE PROC dbo.usp_nextpage
  @anchor AS INT, -- key of last row in prev page
  @n AS INT = 10
AS
SELECT TOP(@n) O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
FROM dbo.Orders AS O
  JOIN dbo.Orders AS A
    ON A.OrderID = @anchor
```

```
    AND (O.OrderDate > A.OrderDate
         OR (O.OrderDate = A.OrderDate
             AND O.OrderID > A.OrderID))
ORDER BY O.OrderDate, O.OrderID;
GO
```

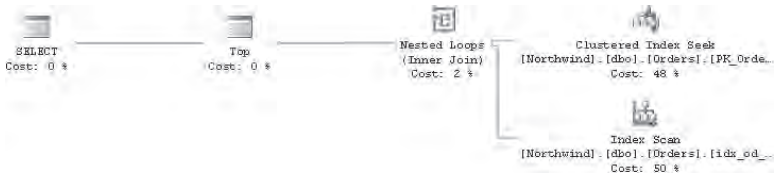
הפרוצדורה מבצעת join על שני מופעים של טבלת הזמנות: אחד נקרא O ומייצג את הדף הבא, ואחד נקרא A ומייצג את העוגן. תנאי ה–join ראשית מסנן את מופע העוגן עם מפתח הקלט, ואז הוא מסנן את המופע המייצג את הדף הבא כך שיוחזרו רק שורות העוקבות אחר העוגן. הטורים OrderDate ו–OrderID קובעים קדימות הן במונחים של ביטויים לוגיים בפסוקית ON המסננת שורות העוקבות אחר העוגן, והן במונחים של הפסוקית ORDER BY עליה נשען TOP. כדי לבחון את הפרוצדורה המאוחסנת, תחילה הרץ אותה עם ה–OrderDate מהשורה האחרונה שהוחזרה מהדף הראשון (10257) כעוגן. בהמשך הרץ אותה שוב עם ה–OrderID של השורה האחרונה בדף השני (10267) כעוגן:

```
EXEC dbo.usp_nextpage @anchor = 10257;
EXEC dbo.usp_nextpage @anchor = 10267;
```

זכור שיישום הלקוח מבצע איטרציה דרך השורות שקיבל חזרה מ– SQL Server, כך שבאופן טבעי הוא יכול לקבל את המפתח מהשורה האחרונה ולהשתמש בו כקלט לקריאה הבאה של הפרוצדורה המאוחסנת.

שתי הקריאות לפרוצדורה הניבו את אותה תוכנית עבודה, המוצגת בתרשים 11–7.

תרשים 11–7: תוכנית עבודה לפרוצדורה המאוחסנת usp_nextpage



בתוכנית תראה פעולת seek יחידה בתוך האינדקס–clustered כדי לקבל את שורת העוגן, ולאחריה סריקה ממוינת בתוך האינדקס–המכסה כדי לקבל את דף השורות הבא. זו אינה תוכנית יעילה ביותר. אידיאלית, ה–optimizer היה מבצע seek בתוך האינדקס–המכסה לשורה הראשונה מדף ההזמנות המבוקש, כשהוא ניגש פיסית רק לשורות הרלוונטיות. הסיבה לקבלת תוכנית בלתי יעילה היא מכיוון שלמסנן יש אופרטור OR בין הביטוי O.OrderDate > A.OrderDate, לבין הביטוי O.OrderDate = A.OrderDate AND O.OrderID > A.OrderID. ראה את הסעיף "טרנספורמציות לוגיות" לפרטים על אופטימיזציה של OR לעומת אופטימיזציה של

AND. לאחר סעיף זה, אביא את היישום היעיל יותר של הפרוצדורה המאוחסנת, המשתמש בלוגיקה של AND.

טרנספורמציות לוגיות

במספר פתרונות שהצגתי, השתמשתי בביטויים לוגיים עם אופרטור OR כדי להתמודד עם קדימות בהתבסס על מספר מאפיינים. כזה היה המקרה בפתרונות האחרונים של דפדוף, התאמת מופעים נוכחיים וקודמים, ובעיות אחרות. השתמשתי בלוגיקה של OR מכיוון שזו הדרך בה רגיל המוח לחשוב. הביטויים הלוגיים המשתמשים בלוגיקה של OR אינטואיטיביים למדי למטרה של קביעת קדימות וזיהוי שורות שיעקבו אחר עוגן מסוים.

עם זאת, בגלל הדרך שבה ה–optimizer של SQL Server עובד, לוגיקת OR היא בעייתית מבחינת ביצועים, במיוחד כאשר חלק מהטורים המסוננים אינם באינדקס. קח לדוגמה את המסנן הבא: col1 = 5 OR col2 = 10. אם יש לך אינדקסים נפרדים על col1 ו–col2, ה–optimizer יכול לסנן את השורות בכל אינדקס ואז לבצע הצלבת אינדקסים בין השניים. אך אם יש לך אינדקס רק על אחד מהטורים, אפילו כאשר המסנן סלקטיבי ביותר, האינדקס חסר תועלת. SQL Server עדיין יצטרך לסרוק את כל הטבלה כדי לראות האם שורות שלא ענו על המסנן הראשון עונות על השני.

מצד שני, ללוגיקת AND יש פוטנציאל ביצועים גבוה בהרבה. עם כל ביטוי, אתה מצמצם את סט התוצאה. שורות המסוננות על ידי אינדקס אחד הן כבר סט–מכיל (superset) של השורות שתחזיר בסופו של דבר. כך שאינדקס על כל אחד מהטורים המסוננים יכול פוטנציאלית להוות יתרון. האם להשתמש באינדקס הקיים או לא זה כבר עניין של סלקטיביות, אך הפוטנציאל קיים. למשל, חשוב על המסנן col1 = 5 AND col2 = 10. האינדקס האופטימלי כאן הוא אינדקס מורכב הנוצר על שני הטורים. עם זאת, אם יש לך אינדקס רק על אחד מהם, והוא סלקטיבי דיו, זה כבר מספיק. SQL Server יכול לסנן את הנתונים דרך אינדקס זה, ואז לחפש את השורות ולבחון האם הן ממלאות גם אחר התנאי השני.

בפרק זה, הביטויים הלוגיים בהם השתמשתי בפתרונות שלי השתמשו בלוגיקת OR כדי לזהות שורות העוקבות אחר עוגן נתון. למשל, נאמר שאתה מביט על השורה בעלת ה– OrderID 11075, ואתה נדרש לזהות את השורות העוקבות, כאשר הקדימות מבוססת על OrderDate ו–OrderID הוא שובר–השוויון. ה–OrderDate של העוגן הוא '19980506'. שאילתה המחזירה את השורות הבאות אחרי שורת עוגן זו היא סלקטיבית ביותר. השתמשתי בלוגיקה הבאה כדי לסנן את השורות הללו:

```
OrderDate > '19980506' OR (OrderDate = '19980506' AND OrderID > 11075)
```

נאמר שאתה יכול לעמוד במחיר של יצירת אינדקס אחד בלבד על OrderDate. אינדקס כזה אינו מספיק בעיני ה–optimizer בשביל לסנן את השורות הרלוונטיות מכיוון שלאחר הביטוי הלוגי המתייחס ל–OrderDate מופיע אופרטור OR, כאשר הצד הימני של האופרטור מתייחס לטורים אחרים (OrderID, במקרה זה). מסנן כזה יניב סריקת טבלה. תוכל לבצע טרנספורמציה לוגית כאן ולקבל ביטוי מקביל המשתמש בלוגיקת AND. להלן הביטוי לאחר טרנספורמציה:

```
OrderDate >= '19980506' AND (OrderDate > '19980506' OR OrderID > 11075)
```

במקום לציין OrderDate > '19980506', אתה מגדיר OrderDate >= '19980506'. כעת באפשרותך להשתמש באופרטור AND ולבקש כל שורה בה OrderDate גדול מה–OrderDate של העוגן (כלומר שה–OrderDate אינו שווה ל–OrderDate של העוגן, ובמקרה זה לא חשוב לך הערך של OrderID); או שה–OrderID גדול מאשר ה–OrderID של העוגן (כלומר שה–OrderDate שווה ל–OrderDate של העוגן). הביטויים הלוגיים מקבילים. אלא שלזה שעבר טרנספורמציה יש את המבנה OrderDate_comparison AND other_logical_expression – כלומר שכעת ניתן לשקול אינדקס על OrderDate בלבד. בשביל להביא מילים אלה לכלל פעולה, ראשית צור טבלה הנקראת MyOrders המכילה את אותם נתונים כמו טבלת Orders, ואינדקס על OrderDate בלבד:

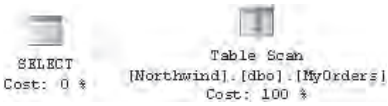
```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO
SELECT * INTO dbo.MyOrders FROM dbo.Orders
CREATE INDEX idx_dt ON dbo.MyOrders(OrderDate);
```

כעת, הרץ את השאילתה בקטע–קוד 7–11, המשתמשת בלוגיקת OR, ובחן את התוכנית בתרשים 7–12.

קטע–קוד 7–11: שאילתה המשתמשת בלוגיקת OR

```
SELECT OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.MyOrders
WHERE OrderDate > '19980506'
   OR (OrderDate = '19980506' AND OrderID > 11075);
```

תרשים 7–12: תוכנית עבודה עבור השאילתה בקטע–קוד 7–11



בתוכנית תראה שקיימת סריקת טבלה, אשר במקרה של טבלה זו עולה 21 קריאות לוגיות. כמובן, תראה משמעותית הרבה יותר I/O, עם גדלי טבלה מציאותיים יותר

כעת, הרץ את השאילתה בקטע–קוד 7–12, המשתמש בלוגיקת AND, ובחן את התוכנית המוצגת בתרשים 7–13.

קטע–קוד 7–12: שאילתה המשתמשת בלוגיקת AND

```
SELECT OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.MyOrders
WHERE OrderDate >= '19980506'
  AND (OrderDate > '19980506' OR OrderID > 11075);
```

תרשים 7–13: תוכנית עבודה עבור השאילתה בקטע–קוד 7–12

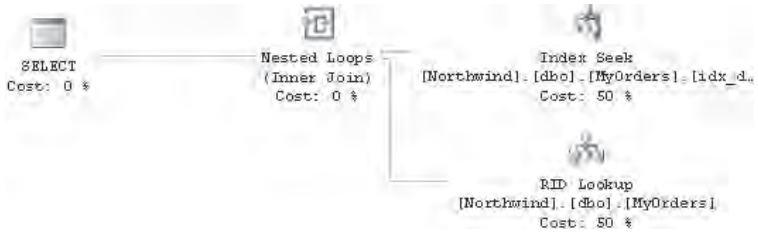
תוכל לראות שיש שימוש באינדקס על OrderDate, ועלות ה–I/O של שאילתה זו היא 6 קריאות לוגיות. יצירת אינדקס על שני הטורים (OrderDate, OrderID) אפילו טוב יותר:

```
CREATE INDEX idx_dt_oid ON dbo.MyOrders(OrderDate, OrderID);
```

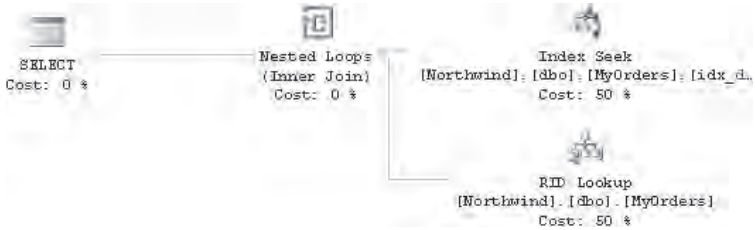
הרץ את השאילתה בקטע–קוד 7–11, המשתמשת בלוגיקת OR. תוכל לראות בתוכנית, המוצגת בתרשים 7–14, שיש שימוש באינדקס החדש. עלות ה–I/O של תוכנית זו היא 6 קריאות לוגיות.

הרץ את השאילתה בקטע–קוד 7–12, המשתמש בלוגיקת AND. תוכל לראות את התוכנית המוצגת בתרשים 7–15, שאולי נראית דומה, אך מניבה אפילו פחות I/O, 4 קריאות לוגיות.

תרשים 7–14: תוכנית עבודה עבור השאילתה בקטע–קוד 7–11, עם האינדקס החדש



תרשים 7–15: תוכנית עבודה עבור השאילתה בקטע–קוד 7–12, עם האינדקס החדש



המסקנה היא, כמובן, ש– SQL Server יכול לבצע אופטימיזציה טובה יותר ללוגיקת AND מאשר ללוגיקת OR. כל הפתרונות שהצגתי בפרק זה יעבדו טוב יותר מבחינת ביצועים, אם תחליף את לוגיקת ה–OR שלהם ללוגיקת AND. בדומה, ייתכן שתוכל להשיג טרנספורמציה כזו עם ביטויים לוגיים אחרים.

מסקנה נוספת היא שכדאי שיהיה אינדקס על כל הטורים הקובעים קדימות. הבעיה היא שבסביבות תפעוליות לא תמיד אתה יכול לעמוד במחיר יצירת אינדקס כזה.

שים לב: כאשר דנים בנושאים המערבים לוגיקה, אני אוהב להשתמש בטבלאות קטנות כמו אלו ב–Northwind, עם נתונים פשוטים ומוכרים. עם טבלאות כאלה, ההפרשים בקריאות לוגיות שאתה מקבל כאשר אתה בוחן את הפתרונות שלך הם קטנים. בבדיקות ובבוחני ביצועים אמיתיים, עליך להשתמש בגדלי טבלאות מציאותיים יותר כנתוני הדוגמה שלך, כמו נתוני הדוגמה בהם השתמשתי בפרק 3. למשל, כאשר תשתמש בפרוצדורה usp_nextpage, המחזירה את דף ההזמנות הבא, תראה הפרשי I/O קטנים מאוד בין לוגיקת OR ולוגיקת AND, כפי שאציג מייד. אך כאשר בחנתי את הפתרון על טבלת Orders בעלת כמיליון שורות, יישום ה–OR עולה יותר מאלף קריאות לוגיות,

בעוד שיישום ה–AND עולה רק 11 קריאות לוגיות, וניגש פיסית רק לדף ההזמנות הרלוונטי.

כשתסיים, אל תשכח להיפטר מטבלת MyOrders שנוצרה עבור דוגמאות אלה:

```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
```

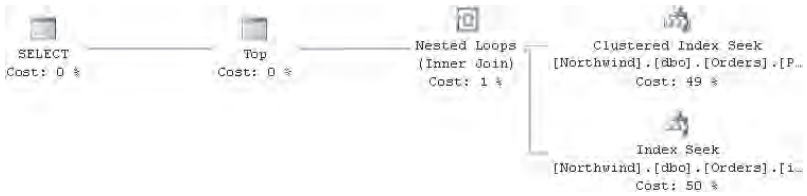
נחזור לפרוצדורה usp_nextpage, להלן היישום היעיל יותר המחליף את לוגיקת ה–OR בלוגיקת AND:

```
ALTER PROC dbo.usp_nextpage
  @anchor AS INT, -- key of last row in prev page
  @n AS INT = 10
AS
SELECT TOP(@n) O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
FROM dbo.Orders AS O
  JOIN dbo.Orders AS A
    ON A.OrderID = @anchor
    AND (O.OrderDate >= A.OrderDate
         AND (O.OrderDate > A.OrderDate
              OR O.OrderID > A.OrderID))
ORDER BY O.OrderDate, O.OrderID;
GO
```

שים לב שביטוי ה–AND בתוך הסוגריים מקביל לוגית לביטוי ה–OR הקודם; רק יישמתי את השיטה שתוארה בסעיף "טרנספורמציות לוגיות". כדי להראות שיישום ה–AND אכן יעיל יותר, הרץ את הקוד הבא ובחן את תוכנית העבודה המוצגת בתרשים 7–16:

```
EXEC dbo.usp_nextpage @anchor = 10257;
```

תרשים 7–16: תוכנית עבודה עבור הפרוצדורה המאוחסנת usp_nextpage – גרסה שנייה



כעת קיבלת את התוכנית הרצויה. אתה רואה פעולת seek יחידה בתוך האינדקס–clustered כדי להשיג את שורת העוגן, ולאחריה seek בתוך האינדקס–המכסה וסריקה חלקית ממוינת, הניגשת פיסית רק לשורות הרלוונטיות בדף ההזמנות הרצוי.

דף קודם

קיימות שתי גישות להתמודדות עם בקשות לדפים קודמים. אחת היא להעלות מקומית ל–cache דפים שנשלחו כבר ללקוח. המשמעות היא שעליך לפתח מכניזם להעלאה ל–cache בלקוח. גישה פשוטה יותר היא ליישם פרוצדורה מאוחסנת אחרת שעובדת כמו הפרוצדורה usp_nextpage ברוורס. פרמטר העוגן יהיה המפתח של השורה הראשונה לאחר הדף בו אתה מעוניין. ההשוואה בתוך הפרוצדורה תשתמש בסימן > במקום בסימן < , והפסוקית TOP תשתמש ברשימת ORDER BY המגדירה את כיוון המיון ההפוך. אם אלו היו השינויים היחידים, היית מקבל את העמוד הנכון, אך בסדר הפוך מהרגיל. בשביל לתקן את הסדר של סט התוצאה, עטוף את השאילתה בטבלה נגזרת, והפעל על טבלה נגזרת זו SELECT ... ORDER BY, עם הסדר הרצוי.

להלן היישום של הפרוצדורה usp_prevpage:

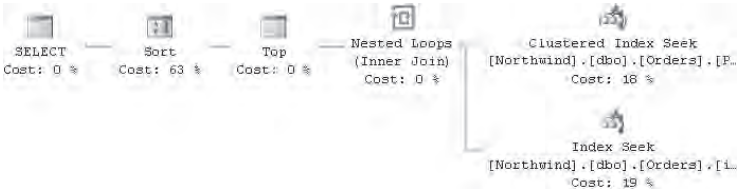
```
CREATE PROC dbo.usp_prevpage
  @anchor AS INT, -- key of first row in next page
  @n AS INT = 10
AS
SELECT OrderID, OrderDate, CustomerID, EmployeeID
FROM (SELECT TOP(@n) O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
      FROM dbo.Orders AS O
        JOIN dbo.Orders AS A
          ON A.OrderID = @anchor
          AND (O.OrderDate <= A.OrderDate
               AND (O.OrderDate < A.OrderDate
                    OR O.OrderID < A.OrderID))
      ORDER BY O.OrderDate DESC, O.OrderID DESC) AS D
ORDER BY OrderDate, OrderID;
GO
```

כדי לבחון את הפרוצדורה, הרץ אותה עם ערכי OrderID מהשורות הראשונות על הדפים שקיבלת כבר:

```
EXEC dbo.usp_prevpage @anchor = 10268;
EXEC dbo.usp_prevpage @anchor = 10258;
```

בחן את התוכנית המוצגת בתרשים 17–7, המופקת עבור ההפעלה של הפרוצדורה usp_prevpage.

תרשים 17–7: תוכנית עבודה עבור הדף הקודם



תמצא תוכנית זהה כמעט לחלוטין לזו שהופקה עבור הפרוצדורה usp_nextpage, עם תוספת של אופרטור Sort, שהוא תוצאה של הפסוקית ORDER BY הנוספת בפרוצדורה usp_prevpage.

כאן רציתי להתמקד בשיטות דפדוף המשתמשות באפשרות TOP. זכור שהנושא מכוסה גם בפרק 4, בו אני מראה פתרונות דפדוף המתבססים על מספרי שורה.

כאשר תסיים, הסר את האינדקס–המכסה עבור פתרונות הדפדוף:

```
DROP INDEX dbo.Orders.idx_od_oid_i_cid_eid;
```

טיפ: כאשר משתמשים בפתרונות כמו אלו בסעיף זה, שינויים בנתוני המקור ישתקפו בבקשות לדפים חדשים. כלומר, אם שורות נוספות נמחקות, או מעודכנות, בקשות לדפים חדשים יישלחו מול הגרסה הנוכחית של הנתונים – שלא לדבר על אפשרות של נפילה מוחלטת של הפרוצדורה, אם מפתח העוגן נעלם. אם התנהגות זו אינה רצויה לך ואתה מעדיף לעבור בין דפים מול מצג קבוע של הנתונים, תוכל ליצור database snapshot ממש לפני שאתה עונה על בקשות דפים. שלח את שאילתות הדפדוף שלך על ה–snapshot, והיפטר מה–snapshot ברגע שהמשתמש מסיים.

לפרטים על database snapshot אנא פנה לספר Inside Microsoft SQL Server 2005: The Storage Engine מאת Kalen Delaney (Microsoft Press, 2006).

שורות רנדומאליות

סעיף זה מכסה תחום אחר של בעיות שבאפשרותך לפתור בעזרת האפשרות TOP – החזרת שורות בצורה רנדומאלית. התמודדות עם רנדומיזאציה ב–T-SQL היא ערמומית למדי. בקשות טיפוסיות לרנדומיזאציה מערבות החזרת שורה רנדומאלית מטבלה, סידור שורות בצורה רנדומאלית, וכדומה. הניסיון הראשון שאתה עשוי לעשות כאשר אתה נדרש להחזיר שורה רנדומאלית, יכול להיות שימוש בפונקציה RAND כלהלן:

```
SELECT TOP(1) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY RAND();
```

עם זאת, אם אתה מנסה להריץ שאילתה זו מספר פעמים, סביר להניח שהתוצאה תאכזב אותך לאחר שתגלה שאינך מקבל שורה רנדומאלית באמת. RAND, כמו מרבית הפונקציות הלא–דטרמיניסטיות האחרות (למשל, GETDATE), מופעלת פעם אחת לכל שאילתה, ולא פעם אחת לכל שורה. כך שבסופו של דבר אתה מקבל אותו ערך של RAND לכל שורה, והפסוקית ORDER BY לא משפיעה על הסדר של סט התוצאה של השאילתה.

טיפ: ייתכן שתופתע לגלות שהפונקציה RAND – כאשר מקבלת זרע (seed) שהוא מספר שלם כקלט – אינה באמת לא–דטרמיניסטית; אלא היא מעין פונקציית hash. בהינתן אותו זרע, RAND(<seed>) תמיד יניב את אותה תוצאה. למשל, הרץ את הקוד הבא מספר פעמים:

```
SELECT RAND(5);
```

תקבל חזרה 0.713666525097956. ואם זה לא מספיק, כאשר אינך מציין זרע, SQL Server אינו בוחר באמת בזרע רנדומאלי. אלא, הזרע החדש מבוסס על ההרצה הקודמת של RAND. לפיכך, הרצת הקוד הבא מספר פעמים יניב את אותן שתי תוצאות (0.713666525097956, ו–0.454560299686459):

```
SELECT RAND(5);
SELECT RAND();
```

השימוש החשוב ביותר של הפונקציה RAND(<seed>) הוא כפי הנראה ליצור נתוני דוגמה הניתנים לייצור–מחדש, מכיוון שאתה יכול לתת לה זרע פעם אחת ואז לקרוא לה שוב ושוב בלי זרע כדי לקבל רצף ערכים מפוזרים היטב.

אם אתה מעוניין בערך רנדומאלי, תזכה להצלחה רבה יותר עם הביטוי הבא:

```
SELECT CHECKSUM(NEWID());
```

שים לב: נראה שלפונקציה NEWID יש מאפייני פיזור טובים; עם זאת, נכון להיום, לא מצאתי כל תיעוד ממיקרוסופט המגדיר שהיא מובטחת או נתמכת.

התנהגות מעניינת של הפונקציה NEWID היא שלא כמו פונקציות לא–דטרמיניסטיות אחרות, NEWID מוערכת בנפרד לכל שורה אם אתה קורא לה בתוך שאילתה. אם תזכור זאת, תוכל לקבל שורה רנדומאלית על ידי שימוש בביטוי הקודם בפסוקית ORDER BY כלהלן:

```
SELECT TOP(1) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY CHECKSUM(NEWID());
```

דבר זה נותן לי הזדמנות להציג דוגמה נוספת לשימוש בפונקציונליות החדשה של TOP, המאפשרת לך להגדיר ביטוי עצמאי כקלט. השאילתה הבאה גם היא מחזירה שורה רנדומאלית:

```
SELECT TOP(1) OrderID, OrderDate, CustomerID, EmployeeID
FROM (SELECT TOP(100e0*(CHECKSUM(NEWID()) + 2147483649)/4294967296e0) PERCENT
        OrderID, OrderDate, CustomerID, EmployeeID
      FROM dbo.Orders
      ORDER BY OrderID) AS D
ORDER BY OrderID DESC;
```

CHECKSUM מחזירה מספר שלם בין 2147483648- ל– 2147483647. הוספת 2147483649 ואז חלוקה בערך הצף 4294967296e0 מניבה מספר רנדומאלי בתחום שבין 0 ל–1 (חוץ מ–0). הכפלת מספר רנדומאלי זה ב–100 מחזיר ערך צף רנדומאלי גדול מ–0 וקטן או שווה ל–100. זכור שהאפשרות TOP PERCENT מקבלת אחוז צף בתחום שבין 0 ל–100, והיא מעגלת מעלה את מספר השורות המוחזרות. אחוז גדול מ–0 מבטיח שתוחזר שורה אחת לפחות. אם כך, השאילתה היוצרת את הטבלה הנגזרת D מחזירה מספר שורות רנדומאלי מהטבלה בהתבסס על מיון OrderID (מפתח ראשי). אז השאילתה החיצונית פשוט מחזירה את השורה האחרונה מהטבלה הנגזרת – כלומר, זו עם ערך ה–OrderID הגדולים ביותר. פתרון זה אינו בהכרח יעיל יותר מאשר הקודם שהצגתי, אך זו הייתה הזדמנות טובה להראות כיצד ניתן להשתמש במאפיינים החדשים של TOP.

עם האופרטור APPLY החדש, באפשרותך כעת למלא בקשות רנדומיזאציה אחרות בצורה קלה ויעילה, ללא צורך להפעיל לוגיקה איטרטיבית ישירות. למשל, השאילתה הבאה מחזירה שלוש הזמנות רנדומאליות לכל עובד:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Employees AS E
  CROSS APPLY
    (SELECT TOP(3) OrderID, CustomerID, OrderDate, RequiredDate
     FROM dbo.Orders AS O
     WHERE O.EmployeeID = E.EmployeeID
     ORDER BY CHECKSUM(NEWID())) AS A;
```

חציון

בפרק 6 בסעיף "פונקציות צבירה מוגדרות–משתמש", דנתי בשיטה לחישוב ערך החציון (median) לכל קבוצה בהתבסס על הפונקציה ROW_NUMBER. כאן אציג שיטות הנשענות על TOP שהן מצד אחד איטיות יותר, אך מצד שני ישימות גם ל– SQL Server 2000. תחילה הרץ את הקוד בקטע–קוד 13–7 ליצירת הטבלה Groups בה השתמשתי בפתרונותיי הקודמים לקבלת החציון.

קטע-קוד 13-7: יצירה ומילוי של טבלת Groups

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Groups') IS NOT NULL
  DROP TABLE dbo.Groups;
GO

CREATE TABLE dbo.Groups
(
  groupid  VARCHAR(10) NOT NULL,
  memberid INT         NOT NULL,
  string   VARCHAR(10) NOT NULL,
  val      INT         NOT NULL,
  PRIMARY KEY (groupid, memberid)
);

INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 3, 'stra1', 6);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 9, 'stra2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 2, 'strb1', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 4, 'strb2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 5, 'strb3', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 9, 'strb4', 11);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 3, 'strc1', 8);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 7, 'strc2', 10);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 9, 'strc3', 12);
GO
```

זכור שחציון הוא הערך האמצעי (בהנחה שהרשימה ממוינת) כאשר הקבוצה היא בעלת מספר אי זוגי של אלמנטים, והוא הממוצע של שני הערכים האמצעיים כאשר היא בעלת מספר זוגי של אלמנטים.

זה תמיד רעיון טוב לטפל בכל מקרה בנפרד, ואז לנסות לראות אם הפתרונות ניתנים למיזוג. אם כך, תחילה הנח שיש מספר אי-זוגי של אלמנטים. תוכל להשתמש בשאילתת

TOP(50) PERCENT כדי לגשת למחצית הראשונה של האלמנטים, כולל האמצעי. זכור שהאפשרות PERCENT מעגלת כלפי מעלה. אז פשוט בצע שאילתה על הערך המקסימלי מסט התוצאה שהוחזר.

כעת טפל במקרה הזוגי. אותה שאילתה בה אתה משתמש לקבלת הערך האמצעי ממספר אי–זוגי של שורות, תניב את הערך הגדול ביותר בחצי הראשון של מספר זוגי של שורות. אז תוכל לכתוב שאילתה דומה כדי להחזיר את הערך הקטן ביותר מהחצי השני. סכם את שני הערכים, חלק בשתיים, ותקבל את החציון במקרה הזוגי.

כעת נסה לגלות האם ניתן למזג את שני הפתרונות. באופן מעניין, הרצת הפתרון עבור המקרה הזוגי מול מספר אי–זוגי של אלמנטים מניב את התוצאה הנכונה, מכיוון ששתי תת–השאילתות המשמשות בפתרון המקרה הזוגי יחזירו את אותה שורה כאשר יש מספר שורות אי–זוגי. הממוצע של שני ערכים שווים הוא כמובן אותו ערך. להלן כיצד נראה הפתרון כאשר ברצונך להחזיר את החציון של הטור val עבור הטבלה כולה:

```
SELECT
  ((SELECT MAX(val)
    FROM (SELECT TOP(50) PERCENT val
          FROM dbo.Groups
          ORDER BY val) AS M1)
   +
   (SELECT MIN(val)
    FROM (SELECT TOP(50) PERCENT val
          FROM dbo.Groups
          ORDER BY val DESC) AS M2))
  /2. AS median;
```

כדי להחזיר את החציון לכל קבוצה, אתה זקוק לשאילתה חיצונית המקבצת את הנתונים לפי groupid. לכל קבוצה, אתה מריץ את חישוב החציון בתת–שאילתה כלהלן:

```
SELECT groupid,
  ((SELECT MAX(val)
    FROM (SELECT TOP(50) PERCENT val
          FROM dbo.Groups AS H1
          WHERE H1.groupid = G.groupid
          ORDER BY val) AS M1)
   +
   (SELECT MIN(val)
    FROM (SELECT TOP(50) PERCENT val
          FROM dbo.Groups AS H2
          WHERE H2.groupid = G.groupid
          ORDER BY val DESC) AS M2))
    /2. AS median
FROM dbo.Groups AS G
GROUP BY groupid;
```

שאילתה זו עובדת ב– SQL Server 2000 עם שני שינויים קלים: ראשית, אתה חייב לכתוב TOP 50 PERCENT במקום TOP(50) PERCENT, ושנית, כדי לעקוף התנהגות בלתי צפויה של שאילתות המשתמשות ב– GROUP BY בצירוף תת–שאילתות, עליך להשתמש ב–SELECT DISTINCT במקום ב– GROUP BY כדי לייצר שורת תוצאה אחת בלבד לכל קבוצה. להלן הפתרון עבור SQL Server 2000:

```
SELECT DISTINCT groupid,
  ((SELECT MAX(val)
    FROM (SELECT TOP 50 PERCENT val
          FROM dbo.Groups AS H1
          WHERE H1.groupid = G.groupid
          ORDER BY val) AS M1)
   +
   (SELECT MIN(val)
    FROM (SELECT TOP 50 PERCENT val
          FROM dbo.Groups AS H2
          WHERE H2.groupid = G.groupid
          ORDER BY val DESC) AS M2))
  /2. AS median
FROM dbo.Groups AS G;
```

סיכום

כפי שבוודאי הבנת מפרק זה, TOP ו–APPLY הם שני מאפיינים שבדרכים רבות משלימים זה את זה. TOP עם היכולות החדשות שלו מאפשר לך כעת לספק ביטויים כקלט והוא נתמך עם משפטי שינויי נתונים. הפונקציונליות החדשה של TOP מחליפה את האפשרות הישנה יותר SET ROWCOUNT. האופרטור החדש APPLY מאפשר שאילתות מאוד מהירות ופשוטות, בהשוואה לחלופות הקודמות, בכל מקרה בו אתה נדרש להפעיל ביטוי טבלה לכל שורה של שאילתה חיצונית.

8

שינוי נתונים

פרק זה מכסה היבטים שונים של שינוי נתונים. אדון בהיבטים לוגיים כגון הוספת שורות חדשות, הסרת שורות כפולות, כמו גם בהיבטי ביצועים כגון התמודדות עם נפחי נתונים גדולים. שים לב שדנתי בהיבטים מסוימים של שינוי נתונים בפרקים אחרים בהם הם התאימו יותר לנושא הפרק. תוכל למצוא הסבר על שינוי נתונים עם TOP בפרק 7 ושל ה– BULK rowset provider, tempdb וטרנזקציות בספר Inside Microsoft SQL Server 2005: T-SQL Programming (Microsoft Press, 2006). ארגנתי פרק זה לפי נושאים, בהתבסס על שלושת הסוגים העיקריים של פעילויות שינוי נתונים: הוספת נתונים, מחיקת נתונים ועדכון נתונים. אכסה פתרונות ב– SQL Server 2000 כמו גם חידושים ב– SQL Server 2005.

הוספת נתונים

בסעיף זה אדון במספר נושאים הקשורים להוספת נתונים, ביניהם: SELECT INTO, INSERT EXEC, הוספת שורות חדשות, INSERT with OUTPUT ומנגנוני רצף.

SELECT INTO

SELECT INTO הוא משפט היוצר טבלה חדשה, המכילה את סט התוצאה של שאילתה, במקום להחזיר את סט התוצאה לשולח. לדוגמה, המשפט הבא יוצר טבלה זמנית הנקראת #MyShippers וממלא אותה בכל השורות מטבלת Shippers ממסד הנתונים Northwind:

```
SELECT ShipperID, CompanyName, Phone
INTO #MyShippers
FROM Northwind.dbo.Shippers;
```

SELECT INTO היא פעולת BULK (להרחבה בנושא, ראה את הסעיף "שיקולי ביצועים אחרים" בסוף הפרק). לפיכך, כאשר מודל ההתאוששות של מסד הנתונים אינו FULL, הפעולה מאוד מהירה בהשוואה לחלופה של יצירת טבלה ואז שימוש ב– INSERT INTO.

טורי הטבלה החדשה יורשים את השמות שלהם, טיפוסי הנתונים, תמיכה ב–NULL ומאפייני IDENTITY, מסט התוצאה של השאילתה. SELECT INTO אינו מעתיק אילוצים, אינדקסים או טריגרים מהמקור של השאילתה. אם אתה זקוק לתוצאות בטבלה עם אותם אינדקסים, אילוצים וטריגרים כמו במקור, תצטרך להוסיף אותם לאחר מכן.

אם אתה צריך העתק מבנה של טבלה כלשהי בלבד, SELECT INTO מאפשר לך להשיג העתק כזה בצורה פשוטה ביותר. אינך צריך לכתוב קוד של משפט CREATE TABLE ולשנות את שם הטבלה. כל שעליך לעשות זה לנפק את המשפט הבא:

```
SELECT * INTO target_table FROM source_table WHERE 1 = 2;
```

ה–optimizer חכם מספיק כדי להבין ששום שורת מקור לא תענה על המסנן 1 = 2. לפיכך, SQL Server לא יטרח לגשת פיסית לנתוני המקור; אלא, הוא ייצור את טבלת היעד בהתבסס על הסכמה של המקור. להלן דוגמה היוצרת טבלה הנקראת MyOrders ב–tempdb, בהתבסס על הסכמה של טבלת Orders ב–Northwind:

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO

SELECT *
INTO dbo.MyOrders
FROM Northwind.dbo.Orders
WHERE 1 = 2;
```

זכור שאם לטור מקור יש את המאפיין IDENTITY, יהיה אותו גם ליעד. למשל, לטור OrderID בטבלת Orders יש את המאפיין IDENTITY. אם אינך רוצה שהמאפיין IDENTITY יועתק לטור היעד, הפעל כל סוג של מניפולציה לטור המקור. למשל, תוכל להשתמש בביטוי OrderID + 0 AS OrderID כלהלן:

```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO

SELECT OrderID+0 AS OrderID, CustomerID, EmployeeID, OrderDate,
  RequiredDate, ShippedDate, ShipVia, Freight, ShipName,
  ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
INTO dbo.MyOrders
FROM Northwind.dbo.Orders
WHERE 1 = 2;
```

במקרה זה, לטור OrderID בטבלת היעד MyOrders אין את המאפיין IDENTITY.

טיפ: נניח שברצונך להכניס את סט התוצאה של פרוצדורה מאוחסנת או של batch דינמי לתוך טבלה חדשה, אך אינך יודע מה הסכמה שעליך ליצור. תוכל להשתמש במשפט SELECT INTO, ולהגדיר OPENQUERY בפסוקית FROM, כשאתה מתייחס לשרת שלך כאילו היה שרת מקושר:

```
EXEC sp_serveroption <your_server>, 'data access', true;
SELECT * INTO <target_table>
FROM OPENQUERY(<your_server>,
  'EXEC {<proc_name> | (<dynamic_batch>)}') AS O;
```

INSERT EXEC

המשפט INSERT EXEC מאפשר לך להפנות סט תוצאה שהתקבל מפרוצדורה מאוחסנת או מ–batch דינמי לתוך טבלה קיימת:

```
INSERT INTO <target_table> EXEC {<proc_name> | (<dynamic_batch>)};
```

משפט זה שימושי מאוד כאשר עליך לשים בצד את סט התוצאה של פרוצדורה מאוחסנת או של batch דינמי לעיבוד נוסף בשרת, בניגוד להחזרת סט התוצאה ללקוח.

אדגים שימושים מעשיים של המשפט INSERT EXEC דרך דוגמה. זכור את הדיון בנושא שיטות דפדוף בפרק 7. סיפקתי פרוצדורה מאוחסנת הנקראת usp_firstpage, המחזירה את דף ההזמנות הראשון בהתבסס על מיון OrderDate, OrderID. סיפקתי גם פרוצדורה מאוחסנת הנקראת usp_nextpage, המחזירה את דף ההזמנות הבא בהתבסס על מפתח קלט (@anchor) המייצג את השורה האחרונה בדף הקודם. בסעיף זה, אשתמש בצורה שונה מעט של הפרוצדורות המאוחסנות, להן אקרא usp_firstrows ו–usp_nextrows. הרץ את הקוד בקטע–קוד 1–8 כדי ליצור את שתי הפרוצדורות.

קטע–קוד 1–8: קוד יצירה של פרוצדורות מאוחסנות המבצעות דפדוף

```
USE Northwind;
GO

-- Index for paging problem
IF INDEXPROPERTY(OBJECT_ID('dbo.Orders'),
     'idx_od_oid_i_cid_eid', 'IndexID') IS NOT NULL
   DROP INDEX dbo.Orders.idx_od_oid_i_cid_eid;
GO
CREATE INDEX idx_od_oid_i_cid_eid
```

```
  ON dbo.Orders(OrderDate, OrderID, CustomerID, EmployeeID);
GO

-- First Rows
IF OBJECT_ID('dbo.usp_firstrows') IS NOT NULL
  DROP PROC dbo.usp_firstrows;
GO
CREATE PROC dbo.usp_firstrows
  @n AS INT = 10 -- num rows
AS
SELECT TOP(@n) ROW_NUMBER() OVER(ORDER BY OrderDate, OrderID) AS RowNum,
  OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate, OrderID;
GO

-- Next Rows
IF OBJECT_ID('dbo.usp_nextrows') IS NOT NULL
  DROP PROC dbo.usp_nextrows;
GO
CREATE PROC dbo.usp_nextrows
  @anchor_rownum  AS INT = 0,   -- row number of last row in prev page
  @anchor_key     AS INT,       -- key of last row in prev page,
  @n              AS INT = 10   -- num rows
AS
SELECT TOP(@n)
  @anchor_rownum
    + ROW_NUMBER() OVER(ORDER BY O.OrderDate, O.OrderID) AS RowNum,
  O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
FROM dbo.Orders AS O
  JOIN dbo.Orders AS A
    ON A.OrderID = @anchor_key
    AND (O.OrderDate >= A.OrderDate
         AND (O.OrderDate > A.OrderDate
              OR O.OrderID > A.OrderID))
ORDER BY O.OrderDate, O.OrderID;
GO
```

שים לב: הפרוצדורות המאוחסנות משתמשות במאפיינים חדשים ב– SQL Server 2005, כך שלא תוכל ליצור אותן ב– SQL Server 2000.

הפרוצדורה המאוחסנת usp_firstrows מחזירה את @n שורות ההזמנות, בהתבסס על מיון OrderDate ו–OrderID. בנוסף לטורים ש–usp_firstpage החזירה, usp_firstrows (כמו גם usp_nextrows) מחזירה גם RowNum, טור המייצג את המיקום הלוגי הגלובלי של השורה בטבלת Orders המלאה לפי המיון שצוין. מכיוון ש–usp_firstrows מחזירה את דף השורות הראשון, RowNum היא פשוט מספר השורה בתוך סט התוצאה.

הפרוצדורה המאוחסנת usp_nextrows מחזירה את @n השורות לאחר שורת עוגן, שהמפתח שלה מסופק כקלט (@anchor_key). עבור שורה בסט התוצאה של usp_nextrows, RowNum שווה למספר השורה הגלובלי של העוגן (@anchor_rownum) ועוד המיקום הלוגי של שורת התוצאה בתוך הסט שהתקבל. אם אינך רוצה שהפרוצדורה המאוחסנת תחזיר מספר שורה גלובלי – אלא, רק את מספר השורה בסט שהתקבל – אל תציין ערך בפרמטר הקלט. במקרה כזה, ברירת המחדל 0 תשמש כמספר השורה של העוגן, ומספר השורה המינימלי שיוקצה יהיה 1.

נניח שברצונך לאפשר למשתמש לבקש כל תחום שורות, מבלי להגביל את הפתרון לדפדוף קדימה–בלבד. ברצונך להימנע גם מסריקה מחדש של חלקים גדולים של הנתונים מטבלת Orders. עליך לפתח מנגנון cache כלשהו, שבו אתה שם בצד העתק של השורות שסרקת כבר, בצירוף מספרי שורה המייצגים את המיקום הלוגי הגלובלי שלהם על פני כל הדפים. כאשר מופיעה בקשה לתחום שורות (דף), אתה בודק קודם האם שורות חסרות ב–cache. במקרה כזה, אתה מוסיף את השורות החסרות ל–cache. אתה אז מבצע שאילתה על ה–cache כדי להחזיר את הדף המבוקש. להלן דוגמה כיצד באפשרותך ליישם פתרון בצד–השרת של מנגנון כזה.

הרץ את הקוד הבא כדי לייצר את הטבלה הזמנית #CachedPages:

```
IF OBJECT_ID('tempdb..#CachedPages') IS NOT NULL
  DROP TABLE #CachedPages;
GO
CREATE TABLE #CachedPages
(
  RowNum      INT NOT NULL PRIMARY KEY,
  OrderID     INT NOT NULL UNIQUE,
  OrderDate   DATETIME,
  CustomerID  NCHAR(5),
  EmployeeID  INT
);
```

לוגיקת ההכנסה ל–cache עטופה בפרוצדורה המאוחסנת usp_getpage, אותה אתה יוצר על ידי הרצת הקוד בקטע–קוד 2–8.

קטע-קוד 8-2: קוד יצירה לפרוצדורה המאוחסנת usp_getpage

```
IF OBJECT_ID('dbo.usp_getpage') IS NOT NULL
  DROP PROC dbo.usp_getpage;
GO
CREATE PROC dbo.usp_getpage
  @from_rownum AS INT,       -- row number of first row in requested page
  @to_rownum   AS INT,       -- row number of last row in requested page
  @rc          AS INT OUTPUT -- number of rows returned
AS

SET NOCOUNT ON;

DECLARE
  @last_key    AS INT,  -- key of last row in #CachedPages
  @last_rownum AS INT,  -- row number of last row in #CachedPages
  @numrows     AS INT;  -- number of missing rows in #CachedPages

-- Get anchor values from last cached row
SELECT @last_rownum = RowNum, @last_key = OrderID
FROM (SELECT TOP(1) RowNum, OrderID
      FROM #CachedPages ORDER BY RowNum DESC) AS D;

-- If temporary table is empty insert first rows to #CachedPages
IF @last_rownum IS NULL
  INSERT INTO #CachedPages
    EXEC dbo.usp_firstrows
      @n = @to_rownum;
ELSE
BEGIN
  SET @numrows = @to_rownum - @last_rownum;

  IF @numrows > 0
    INSERT INTO #CachedPages
      EXEC dbo.usp_nextrows
        @anchor_rownum = @last_rownum,
        @anchor_key    = @last_key,
        @n             = @numrows;
END
```

```
-- Return requested page
SELECT *
FROM #CachedPages
WHERE RowNum BETWEEN @from_rownum AND @to_rownum
ORDER BY RowNum;

SET @rc = @@rowcount;
GO
```

הפרוצדורה המאוחסנת מקבלת את מספרי השורה המייצגים את השורה הראשונה בדף המבוקש (@from_rownum) ואת השורה האחרונה (@to_rownum) כקלטים. מלבד החזרת דף השורות המבוקש, הפרוצדורה המאוחסנת מחזירה גם פרמטר פלט המציין את מספר השורות שהוחזרו (@rc). תוכל לבחון את פרמטר הפלט כדי לקבוע אם הגעת לדף האחרון.

הקוד של הפרוצדורה המאוחסנת מבצע תחילה שאילתה על הטבלה הזמנית #CachedPages כדי לשמור בצד במשתנים המקומיים @last_rownum ו–@last_key את מספר השורה ואת המפתח של השורה האחרונה שנמצאת ב–cache, בהתאמה. אם הטבלה הזמנית ריקה, (@last_rownum IS NULL), הקוד קורא לפרוצדורה usp_firstrows עם משפט INSERT EXEC כדי למלא את #CachedPages בשורות הראשונות עד למספר השורה בגבול הגבוה המבוקש. אם הטבלה הזמנית מכילה כבר שורות, הקוד בודק האם חסרות בה שורות מהדף המבוקש (@to_rownum - @last_rownum > 0). במקרה כזה, הקוד קורא לפרוצדורה usp_nextrows כדי להוסיף את כל השורות החסרות עד למספר השורה בגבול הגבוה המבוקש לטבלה הזמנית.

לבסוף, הקוד מבצע שאילתה על הטבלה הזמנית #CachedPages כדי להחזיר את תחום השורות המבוקש, והוא מאחסן את מספר השורות שהוחזרו בפרמטר הפלט @rc.

כדי לקבל את דף השורות הראשון, בהנחה שגודל הדף הוא 10 שורות, הרץ את הקוד הבא:

```
DECLARE @rc AS INT;

EXEC dbo.usp_getpage
  @from_rownum = 1,
  @to_rownum   = 10,
  @rc          = @rc OUTPUT;

IF @rc = 0
  PRINT 'No more pages.'
ELSE IF @rc < 10
  PRINT 'Reached last page.';
```

תקבל חזרה את 10 השורות הראשונות בהתבסס על מיון OrderDate ו–OrderID. שים לב בקוד שאתה יכול לבחון את פרמטר הפלט כדי לקבוע האם אין יותר דפים (@rc = 0), או האם הגעת לדף האחרון (@rc < 10).

בצע שאילתה על הטבלה הזמנית #CachedPages, ותמצא ש–10 שורות נוספו ל–cache:

```
SELECT * FROM #CachedPages;
```

בקשות נוספות לשורות שהיו כבר ב–cache יסופקו מ–#CachedPages ללא צורך לגשת לטבלת Orders. ביצוע שאילתה על #CachedPages הוא יעיל מאוד, מכיוון שהטבלה מכילה אינדקס–clustered על הטור RowNum. רק לשורות המבוקשות מתבצעת גישה פיסית.

אם תריץ כעת את הקוד הקודם תוך ציון שורות 21 עד 30 כקלטים, הפרוצדורה usp_getpage תוסיף את השורות 11 עד 30 לטבלה הזמנית, ותחזיר את שורות 21 עד 30. בקשות נוספות לשורות עד שורה 30 יסופקו אך ורק מהטבלה הזמנית.

כשתסיים להתנסות עם שיטת דפדוף זו, הרץ את הקוד הבא לצורך ניקוי:

```
IF OBJECT_ID('tempdb..#CachedPages') IS NOT NULL
  DROP TABLE #CachedPages;
GO
IF INDEXPROPERTY(OBJECT_ID('dbo.Orders'),
     'idx_od_oid_i_cid_eid', 'IndexID') IS NOT NULL
  DROP INDEX dbo.Orders.idx_od_oid_i_cid_eid;
GO
IF OBJECT_ID('dbo.usp_firstrows') IS NOT NULL
  DROP PROC dbo.usp_firstrows;
GO
IF OBJECT_ID('dbo.usp_nextrows') IS NOT NULL
  DROP PROC dbo.usp_nextrows;
GO
IF OBJECT_ID('dbo.usp_getpage') IS NOT NULL
  DROP PROC dbo.usp_getpage;
GO
```

הוספת שורות חדשות

הבעיה העומדת במרכז סעיף זה מערבת הוספת שורות מטבלת מקור כלשהי לתוך טבלת יעד, תוך כדי סינון של שורות בעלות מפתחות שאינם קיימים עדיין ביעד בלבד. ייתכן שתיתקל בבעיה זו כאשר עליך לעדכן טבלה ראשית מטבלת תוספות ושינויים – למשל, עדכון data warehouse מרכזי עם מידע מסניפים אזוריים. בסעיף זה, אתמקד בחלק של הבעיה המערב הוספת שורות חדשות.

קיימות כמה שיטות מהן תוכל לבחור. השיטה ההולמת משימה נתונה, במובנים של פשטות וביצועים, תהיה תלויה במספר גורמים. האם טבלת המקור מכילה שורות עם ערכים כפולים בטורים המתייחסים למפתח של טבלת היעד? אם כן, מה הצפיפות שלהן? והאם מובטח שהשורות עם הערכים הכפולים זהות לחלוטין, או שהחלקים הזהים הם רק הטורים המרכיבים את מפתח היעד? ייתכן שהתרחישים השונים שהזכרתי אינם בהירים כרגע, אך אספק פרטים נוספים תוך כדי שאסביר את התרחישים בתוך הקשר.

כדי להדגים שיטות שונות לפתרון המשימה שלפנינו, הרץ את הקוד בקטע–קוד 3–8, היוצר וממלא בנתונים את טבלאות MyOrders, MyCustomers, StageCusts ו–StageOrders.

קטע קוד 3–8: צור וממלא בנתונים טבלאות דוגמה

```
USE tempdb;
GO
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
  DROP TABLE dbo.MyOrders;
GO
IF OBJECT_ID('dbo.MyCustomers') IS NOT NULL
  DROP TABLE dbo.MyCustomers;
GO
IF OBJECT_ID('dbo.StageCusts') IS NOT NULL
  DROP TABLE dbo.StageCusts;
GO
IF OBJECT_ID('dbo.StageOrders') IS NOT NULL
  DROP TABLE dbo.StageOrders;
GO

SELECT *
INTO dbo.MyCustomers
FROM Northwind.dbo.Customers
WHERE CustomerID < N'M';

ALTER TABLE dbo.MyCustomers ADD PRIMARY KEY(CustomerID);

SELECT *
INTO dbo.MyOrders
FROM Northwind.dbo.Orders
WHERE CustomerID < N'M';

ALTER TABLE dbo.MyOrders ADD
  PRIMARY KEY(OrderID),
```

```
  FOREIGN KEY(CustomerID) REFERENCES dbo.MyCustomers;

SELECT *
INTO dbo.StageCusts
FROM Northwind.dbo.Customers;

ALTER TABLE dbo.StageCusts ADD PRIMARY KEY(CustomerID);

SELECT C.CustomerID, CompanyName, ContactName, ContactTitle,
  Address, City, Region, PostalCode, Country, Phone, Fax,
  OrderID, EmployeeID, OrderDate, RequiredDate, ShippedDate,
  ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion,
  ShipPostalCode, ShipCountry
INTO dbo.StageOrders
FROM Northwind.dbo.Customers AS C
  JOIN Northwind.dbo.Orders AS O
  ON O.CustomerID = C.CustomerID;

CREATE UNIQUE CLUSTERED INDEX idx_cid_oid
  ON dbo.StageOrders(CustomerID, OrderID);
ALTER TABLE dbo.StageOrders ADD PRIMARY KEY NONCLUSTERED(OrderID);
```

הבא נתחיל בתרחיש הפשוט ביותר. הרגע ייבאת נתוני לקוח חדשים ומעודכנים לטבלת הביניים StageCusts. כעת עליך להוסיף לטבלת MyCustomers את כל הלקוחות מטבלת StageCusts שאינם קיימים כבר ב–MyCustomers. בנתוני המקור אין לקוחות כפולים. הפתרון הוא להשתמש פשוט בביטוי NOT EXIST כדי לוודא שאתה מוסיף שורות מ–StageCusts עם מפתחות שאינם קיימים עדיין ב–MyCustomers כלהלן:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
  SELECT CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
  FROM dbo.StageCusts AS S
  WHERE NOT EXISTS
    (SELECT * FROM dbo.MyCustomers AS T
     WHERE T.CustomerID = S.CustomerID);
```

כעת נניח שאינך מקבל את טבלת StageCusts; אלא, אתה מקבל טבלת StageOrders המכילה הן נתוני הזמנה והן נתוני לקוח בצורה לא–מנורמלת. לקוח חדש עשוי להופיע במספר שורות בטבלת StageOrders אך צריך להתווסף רק פעם אחת לטבלת MyCustomers. השיטות שיש ברשותך כדי לבודד שורה אחת בלבד לכל לקוח, תלויות בשאלה האם

מובטח שבכל הטורים של הלקוח יהיו כפילויות זהות, או שעשויים להיות הבדלים בטורים שאינם–מפתח (למשל, המבנה של מספרי טלפון). אם מובטח ששורות בעלות אותו CustomerID יהיו בעלות אותם ערכים בכל טורי הלקוח האחרים, תוכל להשתמש בשאילתת NOT EXIST כפי שהצגתי קודם, תוך הוספת פסוקית DISTINCT לטורי הלקוח שאתה מבקש בטבלת StageOrders:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
  SELECT DISTINCT CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
  FROM dbo.StageOrders AS S
  WHERE NOT EXISTS
    (SELECT * FROM dbo.MyCustomers AS T
     WHERE T.CustomerID = S.CustomerID);
```

אם טורי לקוח אחרים מ–CustomerID עשויים להיות שונים בין שורות עם אותו CustomerID, תצטרך לבודד שורה אחת בלבד ללקוח. באופן טבעי, DISTINCT לא יעבוד במקרה כזה מכיוון שהוא מבטל רק שורות כפולות זהות לחלוטין. יתרה מכך, השיטה המשתמשת ב–DISTINCT דורשת סריקה מלאה של טבלת המקור, כך שהיא איטית. תוכל להשתמש במפתח של שורת המקור (OrderID במקרה שלנו) כדי לזהות שורה יחידה לכל לקוח, מכיוון שהמפתח הוא ייחודי. לדוגמה, תוכל להשתמש בתת–שאילתה המחזירה את ה–OrderID המינימלי ללקוח החיצוני:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
   ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
  SELECT CustomerID, CompanyName, ContactName,
   ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
  FROM dbo.StageOrders AS S
  WHERE NOT EXISTS
    (SELECT * FROM dbo.MyCustomers AS T
     WHERE T.CustomerID = S.CustomerID)
    AND S.OrderID = (SELECT MIN(OrderID) FROM dbo.StageOrders AS S2
                     WHERE S2.CustomerID = S.CustomerID);
```

ב– SQL Server 2005 , תוכל להישען על הפונקציה ROW_NUMBER כדי לקבל את הפתרון המהיר ביותר הקיים מבין אלו שהצגתי:

```
INSERT INTO dbo.MyCustomers(CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax)
  SELECT CustomerID, CompanyName, ContactName,
    ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
```

```
  FROM (SELECT
          ROW_NUMBER() OVER(PARTITION BY CustomerID ORDER BY OrderID) AS rn,
          CustomerID, CompanyName, ContactName, ContactTitle, Address, City,
          Region, PostalCode, Country, Phone, Fax
        FROM dbo.StageOrders) AS S
WHERE NOT EXISTS
     (SELECT * FROM dbo.MyCustomers AS T
      WHERE T.CustomerID = S.CustomerID)
    AND rn = 1;
```

השאילתה מחשבת מספרי שורה במחיצות לפי CustomerID, בהתבסס על מיון OrderID, ומבודדת רק שורות עבור לקוחות חדשים, עם מספר שורה השווה ל–1. בצורה זו אתה מבודד לכל לקוח חדש רק את השורה עם ה–OrderID המינימלי.

INSERT עם OUTPUT

SQL Server 2005 מציג תמיכה להחזרת פלט ממשפט שינוי נתונים על ידי פסוקית OUTPUT חדשה. אני אוהב לחשוב על מאפיין זה כ"DML עם תוצאות". הפסוקית OUTPUT נתמכת עבור משפטי INSERT, DELETE ו–UPDATE. בפסוקית OUTPUT, אתה יכול להתייחס לטבלאות המיוחדות inserted ו–deleted. טבלאות מיוחדות אלו מכילות את השורות המושפעות על ידי משפט שינוי הנתונים. אתה משתמש בטבלאות inserted ו–deleted כאן בדומה מאוד לדרך בה אתה משתמש בהן בטריגרים. את הנושא של טריגרים אכסה באריכות בספר **תכנות ב-T-SQL**; כרגע מספיק לומר שהטבלאות inserted ו–deleted בטריגר, מחזיקות את התמונה החדשה והישנה של השורות ששונו על ידי המשפט שהפעיל את הטריגר. עם INSERTs, אתה פונה לטבלה inserted כדי לזהות טורים מהשורות החדשות. עם DELETEs, אתה פונה לטבלה deleted כדי לזהות טורים מהשורות הישנות. עם UPDATEs, אתה פונה לטבלה deleted כדי לזהות את המאפיינים מהשורות המעודכנות לפני השינוי, ואתה פונה לטבלה inserted כדי לזהות את הטורים מהשורות המעודכנות לאחר השינוי. היעד של הפלט יכול להיות הלקוח (יישום לקוח), טבלה, או אפילו שניהם. הדרך הטובה ביותר להסביר מאפיין זה היא דרך דוגמאות. בסעיף זה, אתן דוגמה ל–INSERT, ובהמשך הפרק אספק גם דוגמאות של DELETE ושל UPDATE.

דוגמה למשפט INSERT שבו פסוקית ה–OUTPUT יכולה להיות שימושית מאוד, היא כאשר אתה מנפק INSERT של שורות מרובות לתוך טבלה עם טור identity וברצונך לדעת מה ערכי ה–identity החדשים. עם INSERTs של שורה בודדת, אין בעיה: הפונקציה SCOPE_IDENTITY מספקת את ערך ה–identity האחרון שנוצר על ידי ה–session שלך ב–scope הנוכחי. אך עבור משפט INSERT לשורות מרובות, כיצד אתה מוצא את ערכי identity החדשים? אתה משתמש בפסוקית OUTPUT ומכוון את ערכי identity החדשים חזרה ללקוח או ליעד כלשהו (למשל, משתנה טבלה).

כדי להדגים שיטה זו, ראשית הרץ את הקוד הבא, המייצר את הטבלה CustomersDim:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.CustomersDim') IS NOT NULL
  DROP TABLE dbo.CustomersDim;
GO

CREATE TABLE dbo.CustomersDim
(
  KeyCol      INT          NOT NULL IDENTITY PRIMARY KEY,
  CustomerID  NCHAR(5)     NOT NULL,
  CompanyName NVARCHAR(40) NOT NULL
  /* ... other columns ... */
);
```

תאר לך שטבלה זו מייצגת מימד לקוח ב– data warehouse שלך. כעת עליך להוסיף לטבלת CustomersDim את הלקוחות מ–UK מטבלת Customers במסד הנתונים Northwind. שים לב שליעד יש טור identity הנקרא KeyCol המכיל מפתחות חלופיים עבור לקוחות. לא אכנס כאן לסיבה שמאחורי השימוש הנפוץ במפתחות חלופיים בטבלאות מימדים ב– data warehouse (בניגוד להישענות על מפתחות טבעיים בלבד); זה אינו מוקד הדיון שלי. אני מעוניין רק להדגים שיטה המשתמשת בפסוקית OUTPUT. נניח שלאחר כל insert עליך לבצע עיבוד כלשהו של הלקוחות שזה עתה נוספו ולזהות איזה מפתח חלופי הוקצה לכל לקוח.

הקוד הבא מכריז על משתנה טבלה (@NewCusts), שמפיק משפט INSERT המכניס לקוחות מ–UK לתוך טבלת CustomersDim, מכוון את הערכים החדשים של CustomerID ו–KeyCol לתוך @NewCusts, ומבצע שאילתה על משתנה הטבלה:

```
DECLARE @NewCusts TABLE
(
  CustomerID  NCHAR(5) NOT NULL PRIMARY KEY,
  KeyCol      INT      NOT NULL UNIQUE
);

INSERT INTO dbo.CustomersDim(CustomerID, CompanyName)
    OUTPUT inserted.CustomerID, inserted.KeyCol
    INTO @NewCusts
    -- OUTPUT inserted.CustomerID, inserted.KeyCol
  SELECT CustomerID, CompanyName
  FROM Northwind.dbo.Customers
  WHERE Country = N'UK';

SELECT CustomerID, KeyCol FROM @NewCusts;
```

קוד זה מפיק את הפלט המוצג בטבלה 1–8, בה תוכל לראות את ערכי ה–identity החדשים בטור KeyCol.

טבלה 1-8: תוכן משתנה הטבלה @NewCusts

CustomerID	*KeyCol*
AROUT	1
BSBEV	2
CONSH	3
EASTC	4
ISLAT	5
NORTS	6
SEVES	7

שים לב לפסוקית ה–OUTPUT השנייה המצוינת בקוד כהערה, שאין אחריה פסוקית INTO. הסר את שני המקפים (--) המופיעים לפני הפסוקית אם ברצונך לשלוח גם את הפלט ללקוח; יהיו לך שתי פסוקיות OUTPUT במשפט ה–INSERT.

מנגנוני רצף

מנגנוני רצף מייצרים מספרים בהם אתה משתמש לרוב כמפתחות. SQL Server מספק מנגנון רצף דרך מאפיין הטור IDENTITY. למאפיין IDENTITY קיימות מספר מגבלות שעשויות לגרום לך לחפש מנגנון רצף חלופי. בסעיף זה, אתאר כמה ממגבלות אלו וכן מנגנונים חלופיים ליצירת מפתחות – כמה שמשתמשים במאפיינים מובנים, כמו GUIDs (Global Unique IDentifiers), וכמה שבאפשרותך לפתח בעצמך.

טורי Identity

המאפיין IDENTITY עשוי להיות נוח כאשר ברצונך ש– SQL Server ייצר מפתחות טור בודדים בטבלה. כדי להבטיח ייחודיות, צור PRIMARY KEY או אילוץ UNIQUE על טור ה–identity. כאשר מתבצע INSERT, SQL Server מגדיל את ערך ה–identity של הטבלה ומאחסן אותו בשורה החדשה.

עם זאת, למאפיין IDENTITY קיימות מספר מגבלות שעשויות לגרום לו להיות מנגנון רצף בלתי–מעשי ליישומים מסוימים.

מגבלה אחת היא שהמאפיין IDENTITY הוא תלוי–טבלה. זהו אינו מנגנון רצף עצמאי המקצה ערכים חדשים שבאפשרותך להשתמש בהם בכל צורה שתרצה. תאר לך שעליך לייצר ערכי רצף שישמשו כמפתחות שאינם יכולים להתנגש בין טבלאות.

מגבלה אחרת היא שערך identity מיוצר כאשר מורץ משפט INSERT, לא קודם לכן. עשויים להיות מקרים בהם עליך לייצר את ערך הרצף החדש ואז להשתמש בו במשפט INSERT, ולא בסדר ההפוך.

היבט אחר של המאפיין IDENTITY שיכול להיחשב כמגבלה במקרים מסוימים, הוא שערכי identity מוקצים בצורה בלתי–סינכרונית. כלומר, שמספר sessions המנפיקים inserts מרובי שורות עשויים לקבל ערכי identity בלתי–רציפים. יתרה מכך, ההקצאה של ערך identity חדש אינו חלק מהטרנזקציה שבה נופק ה–INSERT. לעובדות אלו מספר השלכות. SQL Server יגדיל את ערך ה–identity של הטבלה ללא תלות בהצלחה או בכישלון של ה–insert. עלולים להיווצר לך פערים ברצף שלא נוצרו על ידי מחיקה. ישנן מערכות שאינן יכולות לאפשר ערכים חסרים שלא ניתנים להסבר ומעקב (למשל, מערכות הפקת חשבוניות). נסה להסביר למס–הכנסה שכמה מקודי החשבוניות החסרים במערכת שלך הם תוצאה של הדרך שבה מנוהלים ערכי identity.

רצפים מוגדרי–משתמש

אציע זוג פתרונות לבעיה של תחזוקת מנגנון רצף. אדגים פתרון סינכרוני ובלתי–סינכרוני.

יצירת רצף סינכרוני

אתה נזקק למחולל רצף סינכרוני כאשר אתה חייב להיות מסוגל לעקוב אחר כל הערכים ברצף. התרחיש הקלאסי לרצף כזה הוא ייצור מספרי חשבוניות. הדרך להבטיח שלא נוצרים פערים היא לנעול את מקור הרצף כאשר אתה צריך להגדיל אותו ולשחרר את הנעילה רק כאשר הטרנזקציה מסתיימת. אם תחשוב על כך, זוהי בדיוק הדרך בה מתנהגות נעילות בלעדיות כאשר אתה משנה נתונים בטרנזקציה – כלומר, כדי לשנות נתונים נוצרת נעילה, והיא משתחררת כאשר הטרנזקציה מסתיימת (בהצלחה או לאחר rollback). כדי לתחזק רצף כזה, צור טבלה עם שורה בודדת וטור בודד המחזיקה את ערך הרצף האחרון בו היה שימוש. ראשית, שים בטבלה אפס אם ברצונך שהערך הראשון ברצף יהיה 1:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.SyncSeq') IS NOT NULL
  DROP TABLE dbo.SyncSeq;
GO

CREATE TABLE dbo.SyncSeq(val INT);
INSERT INTO dbo.SyncSeq VALUES(0);
```

כעת, כשטבלת הרצף מוכנה, אתאר כיצד אתה משיג ערך רצף יחיד או מקטע של ערכים רציפים בבת אחת.

ערך רצף יחיד

כדי להשיג ערך רצף יחיד, אתה מגדיל את ערך הרצף ב–1 ומחזיר את ערך התוצאה. תוכל להשיג זאת על ידי התחלת טרנזקציה, שינוי ערך הרצף, ואז אחזורו. או שתוכל גם להגדיל את ערך הרצף החדש וגם לאחזרו בפעולה אטומית אחת, על ידי שימוש בתחביר UPDATE מיוחד לצורך העניין. הרץ את הקוד הבא כדי ליצור את הפרוצדורה המאוחסנת המשתמשת בתחביר UPDATE המיוחד ל–T-SQL, המגדיל את ערך הרצף, ומחזיר את הערך החדש כפרמטר פלט:

```
IF OBJECT_ID('dbo.usp_SyncSeq') IS NOT NULL
  DROP PROC dbo.usp_SyncSeq;
GO

CREATE PROC dbo.usp_SyncSeq
  @val AS INT OUTPUT
AS
UPDATE dbo.SyncSeq
  SET @val = val = val + 1;
GO
```

ההקצאה SET@val = val = val + 1 מקבילה ל– SET val = val + 1, @val = val + 1. שים לב ש– SQL Server ראשית ינעל את השורה בלעדית ואז יגדיל את val, יאחזר אותו, וישחרר את הנעילה רק כאשר הטרנזקציה הסתיימה.

בכל עת שאתה צריך ערך רצף חדש, השתמש בקוד הבא:

```
DECLARE @key AS INT;
EXEC dbo.usp_SyncSeq @val = @key OUTPUT;
SELECT @key;
```

כדי לשחרר את הרצף – למשל, כאשר ערך הרצף עומד לחרוג מהגודל המקסימלי – שנה את ערכו לאפס:

```
UPDATE dbo.SyncSeq SET val = 0;
```

בלוק ערכי רצף

אם ברצונך במנגנון להקצאת מקטע של ערכי רצף בבת אחת, עליך לשנות מעט את היישום של הפרוצדורה המאוחסנת כלהלן:

```
ALTER PROC dbo.usp_SyncSeq
  @val AS INT OUTPUT,
  @n   AS INT = 1
```

```
AS
UPDATE dbo.SyncSeq
  SET @val = val + 1, val = val + @n;
GO
```

בארגומנט הנוסף (@n) אתה מציין את גודל הבלוק (כמה ערכי רצף אתה צריך). הפרוצדורה המאוחסנת מגדילה את ערך הרצף הנוכחי ב–@n ומחזירה את הערך הראשון בבלוק דרך פרמטר הפלט @val. פרוצדורה זו מקצה את בלוק ערכי הרצף מ– @val ועד @val + @n - 1.

הקוד הבא מספק דוגמה להקצאה ושימוש בבלוק שלם של ערכי רצף:

```
IF OBJECT_ID('tempdb..#CustsStage') IS NOT NULL
  DROP TABLE #CustsStage
GO

DECLARE @key AS INT, @rc AS INT;

SELECT CustomerID, 0 AS KeyCol
INTO #CustsStage
FROM Northwind.dbo.Customers
WHERE Country = N'UK';

SET @rc = @@rowcount;
EXEC dbo.usp_SyncSeq @val = @key OUTPUT, @n = @rc;

SET @key = @key -1;
UPDATE #CustsStage SET @key = KeyCol = @key + 1;

SELECT CustomerID, KeyCol FROM #CustsStage;
```

שיטה זו היא חלופה לשיטה שהוצגה קודם המשתמשת במאפיין IDENTITY לייצור מפתחות חלופיים ללקוחות מ–UK. קוד זה משתמש במשפט SELECT INTO כדי להוסיף לקוחות מ–UK לתוך טבלה זמנית בשם #CustsStage, תוך שהוא מקצה זמנית 0 כערך של KeyCol בכל שורות היעד. הקוד אז מאחסן את מספר השורות המושפעות (@@rowcount) במשתנה @rc. בשלב הבא, הקוד קורא לפרוצדורה usp_SyncSeq כדי לבקש בלוק ערכי רצף חדשים בגודל @rc. הפרוצדורה המאוחסנת מאחסנת את ערך הרצף הראשון מהבלוק במשתנה @key דרך פרמטר הפלט @val. כעת, הקוד מפחית 1 מ–@key וקורא למשפט UPDATE המיוחד ל–T-SQL כדי להקצות את בלוק ערכי הרצף. ה–UPDATE מבצע מעבר יחיד על השורות ב–#CustsStage. עם כל שורה אותה ה–UPDATE מבקר, הוא מאחסן את הערך של @key + 1 בטור KeyCol וב–@key. המשמעות היא שבכל שורה

חדשה אליה מגיעים, @key גדל באחד ומאוחסן בטור KeyCol. אתה למעשה מפזר את הבלוק החדש של ערכי הרצף בין השורות ב–#CustsStage. אם תריץ את הקוד לאחר ששינית את ערך הרצף ל–0, כפי שהנחתי קודם, #CustsStage יכיל שבעה לקוחות מ–UK, עם ערכי KeyCol בין 1 ל–7. הרץ קוד זה מספר פעמים כדי לראות כיצד אתה מקבל בכל פעם בלוק חדש של ערכי רצף (1 עד 7, 8 עד 14, 15 עד 21 וכך הלאה).

המשפט UPDATE הייחודי ל–T-SQL אינו קיים בתקן ואין ביטחון שייגש לשורות ב–#CustsStage בסדר מסוים. לפיכך, כאשר אתה משתמש בו, אינך יכול לשלוט בסדר שבו SQL Server יקצה את בלוק ערכי הרצף.

קיימת שיטה להקצאת בלוק ערכי הרצף כאשר אתה יכול לשלוט בסדר ההקצאה, אך היא חדשה ל– SQL Server 2005. החלף את משפט ה–UPDATE שזה עתה הוצג ב–UPDATE מול CTE המחשב מספרי שורה בהתבסס על הסדר הרצוי (למשל, CustomerID) כלהלן:

```
WITH CustsStageRN AS
(
  SELECT KeyCol, ROW_NUMBER() OVER(ORDER BY CustomerID) AS RowNum
  FROM #CustsStage
)
UPDATE CustsStageRN SET KeyCol = RowNum + @key;
```

תוכל להשתמש בשיטות דומות כאשר ברצונך רק להקצות רצף של ערכים ייחודיים המתחילים ב–1 (וגדלים ב–1 מכאן והלאה) שאינם קשורים לשום רצף קיים. אתאר שיטות כאלה בהמשך הפרק בסעיף "UPDATE של הצבה".

עד כה, הדגמתי הכנסת סט תוצאה לתוך טבלת יעד ועדכון השורות עם ערכי רצף חדשים. במקום זאת, תוכל להשתמש במשפט SELECT INTO כדי למלא טבלה זמנית עם שורות היעד, וגם כדי להקצות מספרי שורה המתחילים ב–1 וגדלים ב–1 בהמשך (הבה נקרא להם rn). כדי לייצר את מספרי השורה, תוכל להשתמש בפונקציה IDENTITY ב– SQL Server 2000 ובפונקציה ROW_NUMBER ב– SQL Server 2005. מייד לאחר ה–insert, אחסן את הערך @@rowcount במשתנה מקומי (@rc) וקרא לפרוצדורה usp_SyncSeq כדי להגדיל את הרצף ב–@rc ולהקצות את ערך הרצף הראשון בבלוק החדש למשתנה שלך (@key). לבסוף, בצע שאילתה על הטבלה הזמנית, כשאתה מחשב את ערכי הרצף החדשים כ– rn + @key – 1. להלן דוגמת הקוד המלאה המדגימה שיטה זו:

```
IF OBJECT_ID('tempdb..#CustsStage') IS NOT NULL
  DROP TABLE #CustsStage
GO

DECLARE @key AS INT, @rc AS INT;

SELECT CustomerID, IDENTITY(int, 1, 1) AS rn
```

```
  -- In 2005 can use ROW_NUMBER() OVER(ORDER BY CustomerID)
INTO #CustsStage
FROM Northwind.dbo.Customers
WHERE Country = N'UK';

SET @rc = @@rowcount;
EXEC dbo.usp_SyncSeq @val = @key OUTPUT, @n = @rc;

SELECT CustomerID, rn + @key - 1 AS KeyCol FROM #CustsStage;
```

הרץ את הקוד מספר פעמים כדי לראות כיצד אתה מקבל בלוק ערכי רצף חדש בכל פעם.

יצירת רצף בלתי–סינכרוני

מנגנון הרצף הסינכרוני אינו מאפשר פערים, אך הוא עשוי ליצור בעיות בעבודת משתמשים רבים במקביל. זכור שאתה חייב לנעול את הרצף בלעדית כדי להגדיל אותו, ואז עליך לשמור את הנעילה עד סיום הטרנזקציה. ככל שהטרנזקציה ארוכה יותר, כך ארוכה נעילת הרצף. כמובן, פתרון זה עשוי לגרום לתורי עיבודים הממתינים שמקור הרצף ישתחרר, אך אין לך אפשרויות אחרות אם ברצונך לתחזק רצף בצורה סינכרונית. עם זאת, ישנם מקרים בהם לא מפריעים לך פערים ברצף. למשל, נניח שכל שאתה צריך הוא מחולל מפתחות שיבטיח שלא תייצר את אותו מפתח פעמיים. נאמר שאתה צריך את המפתחות הללו לצורך זיהוי ייחודי של שורות בין טבלאות. אינך רוצה שמקור הרצף יינעל לכל משך הטרנזקציה. אלא, אתה רוצה שהרצף יינעל לשבריר שנייה כאשר אתה מגדיל אותו, רק כדי למנוע מעיבודים שונים לקבל את אותו ערך. במילים אחרות, אתה זקוק לרצף בלתי–סינכרוני, כזה שיעבוד מהר הרבה יותר מאשר הסינכרוני, ויאפשר עבודה טובה יותר של משתמשים מרובים במקביל.

אפשרות אחת שתמלא דרישות אלה, היא להשתמש בפונקציות מובנות ש– SQL Server מספק לך כדי לייצר GUIDs (אדון באפשרות זו מייד), אלא ש–GUIDs הם ארוכים (16 בתים). ייתכן שתעדיף להשתמש עבור ערכי הרצף במספרים שלמים, הקטנים יותר משמעותית (4 בתים). כדי להשיג מנגנון רצף בלתי–סינכרוני שכזה, אתה יוצר טבלה (AsyncSeq) עם טור identity כלהלן:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.AsyncSeq') IS NOT NULL
  DROP TABLE dbo.AsyncSeq;
GO

CREATE TABLE dbo.AsyncSeq(val INT IDENTITY(1,1));
```

צור את הפרוצדורה usp_AsyncSeq הבאה, כדי לייצר ערך רצף חדש ולהחזיר אותו דרך פרמטר הפלט @val:

```
IF OBJECT_ID('dbo.usp_AsyncSeq') IS NOT NULL
  DROP PROC dbo.usp_AsyncSeq;
GO

CREATE PROC dbo.usp_AsyncSeq
  @val AS INT OUTPUT
AS
BEGIN TRAN
  SAVE TRAN S1;
  INSERT INTO dbo.AsyncSeq DEFAULT VALUES;
  SET @val = SCOPE_IDENTITY()
  ROLLBACK TRAN S1;
COMMIT TRAN
GO
```

הפרוצדורה פותחת טרנזקציה אך ורק לצורך יצירת נקודת שמירה הנקראת S1. היא מכניסה שורה חדשה ל–AsyncSeq, המייצרת ערך identity חדש בטבלת AsyncSeq ומאחסנת אותו בפרמטר הפלט @val. הפרוצדורה אז מבצעת rollback ל–INSERT. אך rollback אינו מבטל הקצאה של משתנה, והוא גם אינו מבטל את ההגדלה של ערך ה–identity. מלבד זאת, משאב ה–identity אינו נעול במשך כל זמן הפעולה של טרנזקציה חיצונית; אלא, הוא ננעל רק לשבריר שנייה כדי לגדול. התנהגות זו של המאפיין IDENTITY חיונית לתחזוקת רצף בלתי–סינכרוני.

שים לב: נכון לזמן כתיבת מילים אלו, לא מצאתי כל תיעוד רשמי של מיקרוסופט המתאר התנהגות זו של המאפיין IDENTITY.

rollback לנקודת שמירה מבטיח של–rollback לא תהיה כל השפעה על טרנזקציה חיצונית. ה–rollback מונע מטבלת AsyncSeq לגדול. למעשה, היא לעולם לא תכיל שורות שעברו commit מקריאות ל–usp_AsyncSeq.

בכל פעם שאתה צריך את ערך הרצף הבא, הרץ את usp_AsyncSeq, ממש כפי שעשית עם הפרוצדורה הסינכרונית:

```
DECLARE @key AS INT;
EXEC dbo.usp_AsyncSeq @val = @key OUTPUT;
SELECT @key;
```

אלא שהפעם, הרצף לא ייחסם אם אתה מגדיל אותו בתוך טרנזקציה חיצונית. פתרון רצף בלתי–סינכרוני זה, יכול לייצר ערך רצף אחד בלבד בכל פעם.

אם ברצונך לאפס את ערך הרצף, באפשרותך לעשות אחד משני דברים. תוכל להריץ TRUNCATE מול הטבלה, שיאפס את ערך ה–identity:

```
TRUNCATE TABLE dbo.AsyncSeq;
```

או שתוכל להריץ את המשפט DBCC CHECKIDENT עם האפשרות RESEED, כלהלן:

```
DBCC CHECKIDENT('dbo.AsyncSeq', RESEED, 0);
```

Globally Unique Identifiers

SQL Server מספק לך את הפונקציה NEWID, המייצרת GUID (Globally Unique IDentifier) חדש בכל פעם שהיא מופעלת. הפונקציה מחזירה ערך בן 16 בתים המוגדר כ–UNIQUEIDENTIFIER. אם אתה נדרש למנגנון אוטומטי המקצה מפתחות ייחודיים בטבלה, או אפילו על פני טבלאות שונות, באפשרותך ליצור טור UNIQUEIDENTIFIER עם ערך ברירת המחדל NEWID. החיסרון של טור UNIQUEIDENTIFIER המשמש כמפתח הוא שהוא גדול למדי – 16 בתים. דבר זה, כמובן, משפיע על גדלי האינדקסים, ביצועי ה–joins וכן הלאה.

שים לב שהפונקציה NEWID אינה מבטיחה ש–GUID שזה עתה נוצר יהיה גדול מכל GUID שנוצר מוקדם יותר על אותו מחשב. אם אתה זקוק לביטחון כזה, השתמש בפונקציה החדשה NEWSEQUENTIALID, שהוצגה ב– SQL Server 2005 במיוחד למטרה זו.

מחיקת נתונים

בסעיף זה אדון בהיבטים שונים של מחיקת נתונים, ביניהם TRUNCATE לעומת DELETE, הסרת שורות עם נתונים כפולים, DELETE על ידי שימוש ב–joins, DELETEs גדולים ו–DELETE עם OUTPUT.

TRUNCATE לעומת DELETE

אם עליך להסיר את כל השורות מטבלה, השתמש ב– TRUNCATE TABLE ולא ב–DELETE ללא פסוקית WHERE. DELETE תמיד נרשם ללוג בצורה מלאה, ועם טבלאות גדולות הוא עלול לארוך זמן רב למדי. TRUNCATE TABLE תמיד נרשם ללוג בצורה מינימלית, ללא תלות במודל ההתאוששות של מסד הנתונים, ולכן הוא תמיד מהיר יותר משמעותית מאשר DELETE. עם זאת, שים לב ש– TRUNCATE TABLE לא יפעיל כל טריגר של DELETE בטבלה. כדי לתת לך מושג כלשהו לגבי ההבדל, שימוש ב– TRUNCATE TABLE כדי לנקות טבלה בת מיליוני שורות, יכול להיות עניין של שניות, בעוד שניקוי הטבלה עם DELETE יכול לקחת מספר שעות.

טיפ: SQL Server ידחה ניסיונות לבצע DROP TABLE אם קיים אובייקט תלוי–סכמה (schema-bound) המצביע לטבלת היעד. הוא ידחה הן ניסיונות DROP TABLE והן ניסיונות TRUNCATE TABLE אם קיים מפתח זר המצביע לטבלת היעד. מגבלה זו קיימת אפילו כאשר הטבלה הזרה ריקה, ואפילו כאשר המפתח הזר לא פעיל (disabled). אם ברצונך למנוע ניסיונות מקריים של TRUNCATE TABLE ו– DROP TABLE מול טבלאות תפעוליות רגישות, פשוט צור טבלאות דמה עם מפתחות זרים (בלתי פעילים) המצביעים אליהן.

בנוסף להבדל המשמעותי בביצועים בין TRUNCATE TABLE ל–DELETE, קיים גם הבדל בדרך בה הם מטפלים במאפיין IDENTITY. TRUNCATE TABLE מאפס את המאפיין IDENTITY לזרע המקורי שלו, בעוד ש–DELETE לא עושה זאת.

הסרת שורות עם נתונים כפולים

כפילויות לרוב נובעות מאכיפה לא מספיק טובה של אמינות נתונים. אם אינך יוצר מפתח ראשי או אילוץ UNIQUE היכן שהם נדרשים, אתה יכול לצפות שיהיו לך נתונים כפולים בלתי רצויים. אמינות נתונים היא דבר חשוב; אכוף אותה דרך אילוצים ועצב יישומים להגנה על הנתונים שלך.

לאחר שהדגשנו נקודה זו, אם יש לך כפילויות בנתונים ועליך להיפטר מהם, קיימות מספר שיטות בהן באפשרותך להשתמש. האפשרויות ויעילות השיטות תלויות במספר גורמים: לדוגמה מספר הכפילויות ביחס לשורות ייחודיות (צפיפות), האם בטוח שהשורה כולה כפולה, או שאתה יכול להישען רק על סט הטורים שיבנו את המפתח ברגע שיוסרו הכפילויות. למשל, נניח שיש לך טבלה המכילה מידע על הזמנות. לא נוצר מפתח ראשי על הטור OrderID, ומספר ערכי OrderID מופיעים יותר מפעם אחת. עליך לשמור רק שורה אחת לכל ערך OrderID ייחודי.

כדי להדגים שיטות להסרת כפילויות, הרץ את הקוד הבא, המייצר את טבלת OrdersDups וממלא אותה ב– 83,000 שורות המכילות כפילויות רבות:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.OrdersDups') IS NOT NULL
  DROP TABLE dbo.OrdersDups
GO

SELECT OrderID+0 AS OrderID, CustomerID, EmployeeID, OrderDate,
  RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress,
  ShipCity, ShipRegion, ShipPostalCode, ShipCountry
INTO dbo.OrdersDups
FROM Northwind.dbo.Orders, dbo.Nums
WHERE n <= 100;
```

זכור שטבלת Nums היא טבלת עזר של מספרים. דנתי בה בפירוט בפרק 4.

הרץ קוד זה שוב לפני שאתה בוחן כל שיטה כדי שיהיה לך בסיס נתונים זהה בכל הבדיקות שלך.

בטרם אכסה שיטות מבוססות–סטים, ראשית עלי להזכיר שבאפשרותך להשתמש בסמן, הסורק את השורות לפי סדר הטורים הקובעים כפילויות (OrderID במקרה של OrdersDups) ומוחק את כל השורות אותן אתה מזהה ככפילויות. זכור רק שסמן כזה מערב תקורה רבה ולרוב יהיה איטי יותר מאשר פתרונות מבוססי–סטים, במיוחד כאשר צפיפות הכפילויות גבוהה (כפילויות רבות). למרות שעם צפיפות נמוכה של כפילויות, ייתכן שכדאי להשוות את הפתרון מבוסס–הסמן לזה מבוסס–הסטים, במיוחד כאשר אין דרך לזהות שורה בצורה ייחודית (כאשר אין מפתח בטבלה).

הבה נתחיל בשיטה אותה באפשרותך להפעיל כאשר שורות שלמות מוכפלות ויש לך צפיפות גבוהה של כפילויות. השתמש במשפט SELECT DISTINCT ... INTO כדי להעתיק שורות ייחודיות לטבלה חדשה. הסר את הטבלה המקורית, שנה את השם של הטבלה החדשה לשם הטבלה המקורית, ואז צור את כל האילוצים, האינדקסים והטריגרים:

```
SELECT DISTINCT * INTO dbo.OrdersTmp FROM dbo.OrdersDups;
DROP TABLE dbo.OrdersDups;
EXEC sp_rename 'dbo.OrdersTmp', 'OrdersDups';
-- Add constraints, indexes, triggers
```

קוד זה רץ בערך 5 שניות על המערכת שלי מול נתוני הדוגמה שסיפקתי. מה שנחמד בשיטה זו זה שהיא אינה דורשת שיהיה קיים כבר מזהה ייחודי בטבלה.

שיטות אחרות דורשות שיהיה קיים כבר מזהה ייחודי. אם יש לך כבר מזהה כזה, הפתרון שלך לא יכלול את העלות המעורבת בהוספת טור כזה. אם אין לך, תוכל להוסיף טור מזהה למטרה זו, אך הוספת טור כזה עשויה לארוך זמן. אם יש לך כבר טור נומרי קיים שבאפשרותך לדרוס, תוכל להשתמש באחת מהשיטות שאציג בהמשך להקצאת רצף של ערכים לטור קיים. התהליך של דריסת טור נומרי קיים מהיר משמעותית מכיוון שאינו מערב עיצוב מחדש והרחבה של שורות.

כדי לאפשר תהליך הסרת נתונים מהיר, תרצה אינדקס על הטורים הקובעים כפילויות (OrderID) וכן על המזהה הייחודי (קרא לו KeyCol). להלן הקוד שעליך להריץ כדי להוסיף טור identity לטבלת OrdersDups וליצור את האינדקס הרצוי:

```
ALTER TABLE dbo.OrdersDups
  ADD KeyCol INT NOT NULL IDENTITY;
CREATE UNIQUE INDEX idx_OrderID_KeyCol
  ON dbo.OrdersDups(OrderID, KeyCol);
```

כעת השתמש במשפט ה–DELETE הבא כדי להיפטר מהכפילויות:

```
DELETE FROM dbo.OrdersDups
WHERE EXISTS
  (SELECT *
   FROM dbo.OrdersDups AS O2
   WHERE O2.OrderID = dbo.OrdersDups.OrderID
     AND O2.KeyCol > dbo.OrdersDups.KeyCol);
```

משפט זה מוחק את כל ההזמנות שעבורן ניתן למצוא הזמנה אחרת עם OrderID זהה ו–KeyCol גבוה יותר. אם תחשוב על כך, בסופו של דבר תקבל שורה אחת לכל OrderID – זו עם ה–KeyCol הגבוה ביותר. שיטה זו רצה במשך 14 שניות על המערכת שלי, כולל הוספת טור ה–identity ויצירת האינדקס.

יתרון אחד שיש לשיטה זו על פני שיטת ה–DISTINCT הקודמת הוא, שאתה מסתמך רק על הטורים הקובעים כפילויות (OrderID) ועל המפתח החלופי (KeyCol). השיטה עובדת אפילו כאשר טורים אחרים בין השורות המיותרות עם אותו ערך OrderID אינם שווים. עם זאת, שיטה זו עשויה להיות איטית מאוד כאשר קיימת צפיפות כפילויות גבוהה. כדי לשפר את הפתרון בתרחיש של צפיפות–גבוהה, תוכל להשתמש בלוגיקה דומה לזו של הפתרון האחרון; כלומר, שמור שורות עם ה–KeyCol המקסימלי לכל OrderID, אך הכנס שורות ייחודיות אלה לתוך טבלה חדשה על ידי שימוש במשפט SELECT INTO. אז תוכל להיפטר מהטבלה המקורית; לשנות את שם הטבלה החדשה לשם הטבלה המקורי; וליצור מחדש את כל האינדקסים, האילוצים והטריגרים. להלן הקוד המיישם גישה זו, אשר רץ על המערכת שלי במשך 2 שניות בסך הכל:

```
ALTER TABLE dbo.OrdersDups
  ADD KeyCol INT NOT NULL IDENTITY;
CREATE UNIQUE INDEX idx_OrderID_KeyCol
  ON dbo.OrdersDups(OrderID, KeyCol);
GO

SELECT O.OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate,
  ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity,
  ShipRegion, ShipPostalCode, ShipCountry
INTO dbo.OrdersTmp
FROM dbo.OrdersDups AS O
  JOIN (SELECT OrderID, MAX(KeyCol) AS mx
        FROM dbo.OrdersDups
        GROUP BY OrderID) AS U
    ON O.OrderID = O.OrderID
    AND O.KeyCol = U.mx;

DROP TABLE dbo.OrdersDups;
EXEC sp_rename 'dbo.OrdersTmp', 'OrdersDups';
-- Recreate constraints, indexes
```

עד כה ראית מספר פתרונות למחיקת שורות עם ערכים כפולים, והמלצתי על התרחיש בו כל אחד מהם מתאים. אך כולם היו מוגבלים בצורה זו או אחרת. גישת ה–DISTINCT דורשת שוויון של שורות שלמות בין כפילויות, ושתי השיטות האחרות דורשות מזהה ייחודי בטבלה. ב– SQL Server 2005, באפשרותך להשתמש ב–CTE ובפונקציה ROW_NUMBER כדי לייצר פתרון מהיר ללא חסרונות אלה:

```
WITH Dups AS
(
  SELECT *,
    ROW_NUMBER() OVER(PARTITION BY OrderID ORDER BY OrderID) AS rn
  FROM dbo.OrdersDups
)
DELETE FROM Dups WHERE rn > 1;
```

השאילתה המגדירה את ה– CTE Dups מייצרת מספרי שורה המתחילים ב–1 לכל מחיצה של שורות בעלות אותו OrderID, כלומר שלכל סט שורות בעלות אותו ערך OrderID יוקצו מספרי שורה המתחילים מ–1 באופן עצמאי מהמחיצות האחרות. כל מספר שורה כאן מייצג את מספר הכפילות. הפונקציה ROW_NUMBER דורשת ממך לציין פסוקית ORDER BY, אפילו כאשר לא ממש חשוב לך כיצד מספרי השורה מוקצים בתוך כל מחיצה. באפשרותך לציין את אותו טור בו אתה משתמש בפסוקית PARTITION BY (OrderID) גם בפסוקית ORDER BY. לפסוקית ORDER BY כזו לא תהיה כל השפעה על הקצאת מספרי שורה בתוך כל מחיצה. חשוב מכך, בעוד שמספור השורות אינו דטרמיניסטי, תהיה בדיוק שורה אחת בכל מחיצה שבה rn שווה 1.

לבסוף, השאילתה החיצונית פשוט מוחקת שורות להן מספר כפילויות גדול מ–1 על ידי ה–CTE, ומשאירה שורה אחת בלבד לכל ערך OrderID.

פתרון זה רץ שנייה אחת בלבד על המערכת שלי; הוא אינו דורש מזהה ייחודי בטבלה; והוא מאפשר לך לזהות כפילויות בהתבסס על כל טור או טורים שאתה בוחר.

DELETE על ידי שימוש ב–Joins

T-SQL תומך בתחביר ייחודי עבור DELETE ו–UPDATE בהתבסס על joins. כאן אסביר על DELETEs בהתבסס על joins, ובהמשך, בסעיף UPDATE, אסביר על UPDATEs בהתבסס על joins.

שים לב: תחביר זה אינו קיים בתקן, ויש להימנע ממנו אלא אם כן יש יתרון משמעותי על פני תחביר התקן, כפי שאתאר בסעיף זה.

ראשית אתאר את התחביר, ואז אציג דוגמאות בהן הוא מספק פונקציונליות שאינה קיימת בתחביר התקן.

אתה כותב DELETE בהתבסס על join באופן דומה לזה בו אתה כותב SELECT בהתבסס על join. אתה מחליף את פסוקית ה–SELECT ב– DELETE FROM <target_table>, כאשר <target_table> היא הטבלה ממנה ברצונך למחוק שורות. שים לב שעליך לציין את כינוי הטבלה, אם כינוי כזה צוין.

השימוש הטיפוסי למאפיין זה הוא הקלה על מחיקת שורות הממלאות תנאי EXISTS או NOT EXISTS, כדי להימנע מהצורך לציין תת–שאילתה עבור התנאי המתאים פעמיים. ישנם אנשים שאוהבים גם את העובדה שהוא מאפשר לך לכתוב שאילתת SELECT, ואז לשנות את ה–SELECT ל–DELETE.

כדי להדגים עד כמה שאילתת join של SELECT דומה לשאילתת join של DELETE, להלן שאילתה המחזירה פרטי הזמנה עבור הזמנות שבוצעו ב–6 במאי 1998 או מאוחר יותר:

```
USE Northwind;

SELECT OD.*
FROM dbo.[Order Details] AS OD
  JOIN dbo.Orders AS O
    ON OD.OrderID = O.OrderID
WHERE O.OrderDate >= '19980506';
```

אם ברצונך למחוק פרטי הזמנה עבור הזמנות שבוצעו ב–6 במאי 1998 או לאחר מכן, פשוט החלף את SELECT OD.* בשאילתה הקודמת ב– DELETE FROM OD:

```
BEGIN TRAN

DELETE FROM OD
FROM dbo.[Order Details] AS OD
  JOIN dbo.Orders AS O
    ON OD.OrderID = O.OrderID
WHERE O.OrderDate >= '19980506';

ROLLBACK TRAN
```

בכמה מהדוגמאות שלי, אני משתמש בטרנזקציה ומבצע rollback לשינוי, כך שניתן לנסות את הדוגמאות ללא שינוי קבוע של הטבלאות לדוגמה. שאילתת DELETE ספציפית זו שאינה בתקן יכולה להיכתב כשאילתה העומדת בתקן על ידי שימוש בתת–שאילתה:

```
BEGIN TRAN

DELETE FROM dbo.[Order Details]
```

```
WHERE EXISTS
  (SELECT *
   FROM dbo.Orders AS O
   WHERE O.OrderID = dbo.[Order Details].OrderID
     AND O.OrderDate >= '19980506');

ROLLBACK TRAN
```

במקרה זה, ל–DELETE שאינו בתקן אין כל יתרון על פני זה שבתקן – לא מבחינת ביצועים ולא מבחינת פשטות, כך שאיני רואה כל סיבה להשתמש בו. עם זאת, תמצא מקרים בהם קשה להסתדר ללא שימוש בתחביר הייחודי. לדוגמה, נניח שעליך למחוק ממשתנה טבלה, ועליך להתייחס למשתנה הטבלה מתוך תת–שאילתה. T-SQL אינו תומך בציון שם משתנה טבלה כתחילית לשם טור.

הקוד הבא מכריז על משתנה טבלה הנקרא @MyOD ושם בו מספר פרטי הזמנה, המזוהים על ידי (OrderID, ProductID). הקוד אז מנסה למחוק מ–@MyOD את כל השורות בעלות מפתחות המופיעים כבר בטבלת פרטי הזמנה:

```
DECLARE @MyOD TABLE
(
  OrderID   INT NOT NULL,
  ProductID INT NOT NULL,
  PRIMARY KEY(OrderID, ProductID)
);

INSERT INTO @MyOD VALUES(10001, 14);
INSERT INTO @MyOD VALUES(10001, 51);
INSERT INTO @MyOD VALUES(10001, 65);
INSERT INTO @MyOD VALUES(10248, 11);
INSERT INTO @MyOD VALUES(10248, 42);

DELETE FROM @MyOD
WHERE EXISTS
  (SELECT * FROM dbo.[Order Details] AS OD
   WHERE OD.OrderID = @MyOD.OrderID
     AND OD.ProductID = @MyOD.ProductID);
```

קוד זה נכשל ומניב את הודעת השגיאה הבאה:

```
Msg 137, Level 15, State 2, Line 17
Must declare the scalar variable "@MyOD".
```

הסיבה לכישלון היא ש–T-SQL לא תומך בציון שם משתנה טבלה כתחילית לשם טור. יתרה מכך, T-SQL אינו מאפשר לך לתת כינוי לטבלת היעד ישירות; אלא, הוא דורש ממך לעשות זאת דרך פסוקית FROM שנייה כלהלן:

```
DELETE FROM MyOD
FROM @MyOD AS MyOD
WHERE EXISTS
  (SELECT * FROM dbo.[Order Details] AS OD
   WHERE OD.OrderID = MyOD.OrderID
     AND OD.ProductID = MyOD.ProductID);
```

שים לב: אם ברצונך לבחון קוד זה, ודא שאתה מריץ אותו מייד לאחר הכרזה על משתנה טבלה ומילויו באותו batch. אחרת, תקבל שגיאה האומרת שהמשתנה @MyOD לא הוכרז. כמו כל משתנה אחר, התחום של משתנה טבלה הוא ה–batch המקומי.

פתרון אחר הוא להשתמש ב–join במקום בתת–השאילתה, אשר בו באפשרותך לתת כינויים גם לטבלאות:

```
DELETE FROM MyOD
FROM @MyOD AS MyOD
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = MyOD.OrderID
  AND OD.ProductID = MyOD.ProductID;
```

ב– SQL Server 2005, ניתן להשתמש ב–CTE כאלטרנטיבה למתן כינוי למשתנה הטבלה, דבר שמאפשר פתרון פשוט יותר:

```
WITH MyOD AS (SELECT * FROM @MyOD)
DELETE FROM MyOD
WHERE EXISTS
  (SELECT * FROM dbo.[Order Details] AS OD
   WHERE OD.OrderID = MyOD.OrderID
     AND OD.ProductID = MyOD.ProductID);
```

CTEs שימושיים ביותר בתרחישים אחרים, בהם עליך לשנות נתונים בטבלה אחת בהתבסס על נתונים אותם אתה בוחן בטבלה אחרת. הדבר מאפשר לך לפשט את הקוד שלך ובמקרים רבים, להימנע מהסתמכות על משפטי שינוי נתונים המשתמשים ב–joins.

DELETE עם OUTPUT

בפרק 7 תיארתי שיטה למחיקת נפחי נתונים גדולים מטבלה קיימת במקטעים, כדי להימנע מבעיות של פיצוץ לוג והסלמת נעילות. כאן אראה כיצד ניתן להשתמש בפסוקית OUTPUT החדשה, כדי לשמור בארכיון נתונים שאתה מוחק. כדי להדגים את השיטה, ראשית הרץ את הקוד הבא, אשר יוצר את טבלת LargeOrders וממלא אותה במעט יותר משני מיליון הזמנות שבוצעו בשנים 2000 עד 2006:

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.LargeOrders') IS NOT NULL
  DROP TABLE dbo.LargeOrders;
GO
SELECT IDENTITY(int, 1, 1) AS OrderID, CustomerID, EmployeeID,
  DATEADD(day, n-1, '20000101') AS OrderDate,
  CAST('a' AS CHAR(200)) AS Filler
INTO dbo.LargeOrders
FROM Northwind.dbo.Customers AS C,
  Northwind.dbo.Employees AS E,
  dbo.Nums
WHERE n <= DATEDIFF(day, '20000101', '20061231') + 1;

CREATE UNIQUE CLUSTERED INDEX idx_od_oid
  ON dbo.LargeOrders(OrderDate, OrderID);

ALTER TABLE dbo.LargeOrders ADD PRIMARY KEY NONCLUSTERED(OrderID);
```

שים לב: הרצת הקוד אמורה להימשך מספר דקות, והיא תדרוש נפח בסביבות ה–1GB במסד הנתונים tempdb שלך. כמו כן, הקוד מתייחס לטבלת העזר Nums, בה דנתי בפרק 4.

כתזכורת, השתמש בשיטה הבאה כדי למחוק את כל השורות בעלות OrderDate גבוה מ–2001 במקטעים של 5000 שורות (אך אל תריץ אותה עדיין):

```
WHILE 1 = 1
BEGIN
  DELETE TOP (5000) FROM dbo.LargeOrders WHERE OrderDate < '20010101';
  IF @@rowcount < 5000 BREAK;
END
```

זכור שב– SQL Server 2005 באפשרותך להשתמש ב– DELETE TOP במקום האפשרות הישנה של SET ROWCOUNT. מוקדם יותר בפרק, הצגתי את התמיכה של SQL Server 2005 בפסוקית OUTPUT החדשה, המאפשרת לך להחזיר פלט ממשפט המשנה נתונים. זכור שבאפשרותך להפנות את הפלט לתוך טבלה זמנית או קבועה, לתוך משתנה טבלה, או חזרה לשולח. הראיתי דוגמה המשתמשת במשפט INSERT, וכאן אראה כזו המשתמשת במשפט DELETE. נניח שהיית רוצה לשפר את הפתרון המוחק נתונים היסטוריים במקטעים, כך שגם ישמור בארכיון את הנתונים שאתה מוחק. הרץ את הקוד הבא ליצירת הטבלה OrdersArchive, בה תאחסן את ההזמנות בארכיון:

```
CREATE TABLE dbo.OrdersArchive
(
  OrderID     INT        NOT NULL PRIMARY KEY NONCLUSTERED,
  CustomerID  NCHAR(5)   NOT NULL,
  EmployeeID  INT        NOT NULL,
  OrderDate   DATETIME   NOT NULL,
  Filler      CHAR(200)  NOT NULL
);

CREATE UNIQUE CLUSTERED INDEX idx_od_oid
  ON dbo.OrdersArchive(OrderDate, OrderID);
GO
```

על ידי שימוש בפסוקית OUTPUT החדשה, תוכל לכוון את השורות המחוקות מכל מקטע לתוך טבלת OrdersArchive. לדוגמה, הקוד הבא הוא הפתרון המשופר, המוחק שורות בעלות OrderDate קודם ל–2001 במקטעים וגם שומר אותן בארכיון:

```
WHILE 1=1
BEGIN
  BEGIN TRAN
    DELETE TOP(5000) FROM dbo.LargeOrders
      OUTPUT deleted.* INTO dbo.OrdersArchive
    WHERE OrderDate < '20010101';

    IF @@rowcount < 5000
    BEGIN
      COMMIT TRAN
      BREAK;
    END
  COMMIT TRAN
END
```

שים לב: קוד זה ירוץ במשך מספר שניות.

טבלת OrdersArchive כעת מחזיקה הזמנות ארכיון שבוצעו לפני 2001.

שים לב: כאשר משתמשים בפסוקית OUTPUT כדי להפנות את הפלט לתוך טבלה, לטבלה לא יכולים להיות טריגרים פעילים או אילוצי CHECK, היא גם אינה יכולה להשתתף בשום צד של אילוץ מפתח זר. אם טבלת היעד אינה עומדת בדרישות אלו, באפשרותך להפנות את הפלט לתוך טבלה זמנית או לתוך משתנה טבלה, ואז להעתיק את השורות משם לטבלת היעד.

ישנם יתרונות חשובים לשימוש בפסוקית OUTPUT כאשר ברצונך לשמור בארכיון נתונים שאתה מוחק. ללא הפסוקית OUTPUT, עליך לבצע ראשית שאילתה על הנתונים כדי לשמור אותם בארכיון, ואז למחוק אותם. שיטה כזו איטית ומורכבת יותר. בשביל להבטיח ששורות חדשות המתאימות למסנן לא יתווספו בין ה–SELECT ל–DELETE (ידועות גם כפאנטומים), אתה נדרש לנעול את הנתונים שאתה שומר בארכיון על ידי שימוש ב– serializable isolation level. עם הפסוקית OUTPUT, לא רק שתקבל ביצועים טובים יותר, אלא שגם לא תצטרך לחשוש מפאנטומים, שכן מובטח לך שתקבל בדיוק מה שמחקת חזרה מהפסוקית OUTPUT.

עדכון נתונים

סעיף זה כולל מספר היבטים של עדכון נתונים, ביניהם UPDATEs על ידי שימוש ב–joins, UPDATE עם OUTPUT, ומשפטי SELECT ו–UPDATE המבצעים הצבות למשתנים.

UPDATE על ידי שימוש ב–joins

מוקדם יותר בפרק, הזכרתי ש–T-SQL תומך בתחביר שאינו בתקן עבור שינוי נתונים בהתבסס על join, והראיתי דוגמאות DELETE. כאן אעסוק ב–UPDATEs המתבססים על joins, ואתמקד במקרים בהם לתחביר שאינו בתקן יש יתרונות על פני התחביר הנתמך בתקן. אראה ש– SQL Server 2005 מציג חלופות פשוטות יותר שלמעשה מבטלות את הצורך בתחביר ה–UPDATE הישן המשתמש ב–joins.

אתחיל באחד המקרים בהם ל–UPDATE המתבסס על join היו יתרונות מבחינת ביצועים, על פני ה–UPDATE שבתקן הנתמך על ידי T-SQL. נניח שהיית רוצה לעדכן את מידע המשלוח עבור הזמנות שבוצעו על ידי לקוחות מ–USA, כך שתדרוס את הטורים ShipCountry, ShipRegion ו–ShipCity עם ערכי הטורים Country, Region ו–City מטבלת Customers. תוכל להשתמש בתת–שאילתה אחת לכל אחד מערכי הטור החדשים, ועוד אחת בפסוקית WHERE, כדי לסנן הזמנות שבוצעו על ידי לקוחות מ–USA כלהלן:

```
USE Northwind;

BEGIN TRAN

  UPDATE dbo.Orders
    SET ShipCountry  = (SELECT C.Country FROM dbo.Customers AS C
                        WHERE C.CustomerID = dbo.Orders.CustomerID),
        ShipRegion   = (SELECT C.Region FROM dbo.Customers AS C
                        WHERE C.CustomerID = dbo.Orders.CustomerID),
        ShipCity     = (SELECT C.City FROM dbo.Customers AS C
                        WHERE C.CustomerID = dbo.Orders.CustomerID)
   WHERE CustomerID IN
     (SELECT CustomerID FROM dbo.Customers WHERE Country = 'USA');

ROLLBACK TRAN
```

שוב, אני מבצע rollback לטרנזקציה כך שהשינוי לא ישפיע על מסד הנתונים Northwind. על אף שהיא תואמת לתקן, שיטה זו איטית מאוד. כל תת–שאילתה כזו מערבת גישה נפרדת להחזרת הטור המבוקש מטבלת Customers. רציתי לספק תרשים עם תוכנית העבודה הגרפית עבור UPDATE זה, אך היא פשוט גדולה מדי! בקש תוכנית עבודה גרפית ב–SSMS ותראה במו עיניך.

תוכל לכתוב UPDATE בהתבסס על join, כדי לבצע את אותה משימה כלהלן:

```
BEGIN TRAN

  UPDATE O
    SET ShipCountry = C.Country,
        ShipRegion = C.Region,
        ShipCity = C.City
  FROM dbo.Orders AS O
    JOIN dbo.Customers AS C
      ON O.CustomerID = C.CustomerID
  WHERE C.Country = 'USA';

ROLLBACK TRAN
```

קוד זה קצר ופשוט יותר, וה–optimizer מייצר תוכנית יעילה יותר עבורו, כפי שתוכל לראות אם תבקש את תוכנית העבודה הגרפית ב–SSMS. תמצא בתוכנית העבודה שטבלת Customers נסרקת פעם אחת בלבד, ודרך סריקה זו, ה– query processor ניגש לכל טורי הלקוח להם הוא נדרש. תוכנית זו מדווחת מחצית מעלות ההפעלה המשוערת של התוכנית הקודמת. בפועל, אם תשווה את שני הפתרונות מול טבלאות גדולות יותר,

תמצא שהבדלי הביצועים גבוהים הרבה יותר. חבל רק ששיטת ה–UPDATE עם join אינה נתמכת בתקן.

ANSI תומך בתחביר הנקרא row value constructors המאפשר לך לפשט שאילתות כמו זו שזה עתה הוצגה. תחביר זה מאפשר לך לציין וקטורים של טורים וביטויים ומבטל את הצורך לנפק תת–שאילתה לכל טור בנפרד. הדוגמה הבאה מציגה תחביר זה:

```
UPDATE dbo.Orders
  SET (ShipCountry, ShipRegion, ShipCity) =
    (SELECT Country, Region, City
     FROM dbo.Customers AS C
     WHERE C.CustomerID = dbo.Orders.CustomerID);
     WHERE CustomerID IN
(SELECT CustomerID FROM dbo. Customers WHERE Country =  'USA' );
```

עם זאת, T-SQL עדיין לא תומך ב– row value constructors. תמיכה כזו תאפשר פתרונות פשוטים יותר העומדים בתקן ותאפשר כמובן אופטימיזציה טובה. אף על פי כן, עוד יש תקווה. על ידי שימוש ב–CTE, תוכל למצוא פתרון פשוט המניב תוכנית יעילה דומה מאוד לזו המשתמשת ב–UPDATE מבוסס–join. פשוט צור CTE מה–join, ואז עדכן את ה–CTE כלהלן:

```
BEGIN TRAN;

WITH UPD_CTE AS
(
  SELECT
    O.ShipCountry  AS set_Country, C.Country AS get_Country,
    O.ShipRegion   AS set_Region,  C.Region  AS get_Region,
    O.ShipCity     AS set_City,    C.City    AS get_City
  FROM dbo.Orders  AS O
    JOIN dbo.Customers AS C
      ON O.CustomerID = C.CustomerID
  WHERE C.Country = 'USA'
)
UPDATE UPD_CTE
  SET set_Country = get_Country,
      set_Region  = get_Country,
      set_City    = get_City;

ROLLBACK TRAN
```

שים לב: על אף ש–CTEs מוגדרים על ידי ANSI SQL:1999, תחביר ה–DELETE וה–UPDATE מול CTEs המיושם ב– SQL Server 2005 אינו נתמך על ידי התקן.

UPDATE זה מייצר תוכנית זהה לזו שנוצרה עבור ה–UPDATE בהתבסס על join.

ישנה סוגיה נוספת לה עליך להיות מודע כאשר אתה משתמש ב–UPDATE מבוסס–join. כאשר אתה משנה את הטבלה בצד ה"רבים" של join אחד–לרבים, אתה עשוי לקבל עדכון לא–דטרמיניסטי. כדי להדגים את הבעיה, הרץ את הקוד הבא, היוצר את טבלאות Customers ו–Orders וממלא אותן בנתונים לדוגמה:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
GO

CREATE TABLE dbo.Customers
(
  custid VARCHAR(5) NOT NULL PRIMARY KEY,
  qty    INT        NULL
);

INSERT INTO dbo.Customers(custid) VALUES('A');
INSERT INTO dbo.Customers(custid) VALUES('B');

CREATE TABLE dbo.Orders
(
  orderid INT        NOT NULL PRIMARY KEY,
  custid  VARCHAR(5) NOT NULL REFERENCES dbo.Customers,
  qty     INT        NOT NULL
);

INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(1, 'A', 20);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(2, 'A', 10);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(3, 'A', 30);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(4, 'B', 35);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(5, 'B', 45);
INSERT INTO dbo.Orders(orderid, custid, qty) VALUES(6, 'B', 15);
```

קיים יחס אחד–לרבים בין Customers ל–Orders. שים לב שלכל שורה ב–Customers יש כרגע שלוש שורות קשורות ב–Orders. כעת, בחן את ה–UPDATE הבא וראה אם באפשרותך לנחש כיצד טבלת Customers תיראה לאחר ה–UPDATE:

```
UPDATE Customers
  SET qty = O.qty
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON C.custid = O.custid;
```

האמת היא שה–UPDATE אינו דטרמיניסטי. אינך יכול להבטיח אילו מהערכים משורות Orders המקושרות ישמש לעדכן את הערך qty ב–Customers. זכור שאינך יכול להניח או להישען על כל סדר פיסי של הנתונים. לדוגמה, הרץ את השאילתה הבאה מול Customers לאחר שהרצת את ה–UPDATE הקודם:

```
SELECT custid, qty FROM dbo.Customers;
```

ייתכן שתקבל את הפלט המוצג בטבלה 2–8.

טבלה 2–8: תוכן אפשרי של Customers לאחר UPDATE לא–דטרמיניסטי

custid	*qty*
A	20
B	35

אך באותה מידה, אתה עשוי לקבל את הפלט המוצג בטבלה 3–8.

טבלה 3–8: תוכן אפשרי נוסף של Customers לאחר UPDATE לא–דטרמיניסטי

custid	*qty*
A	10
B	15

כשתסיים להתנסות ב–UPDATEs לא–דטרמיניסטיים, הרץ את הקוד הבא כדי להסיר את הטבלאות Orders ו–Customers:

```
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
```

UPDATE עם OUTPUT

כמו עם משפטי INSERT ו–DELETE, משפטי UPDATE תומכים גם בפסוקית OUTPUT, המאפשרת לך להחזיר פלט כאשר אתה מעדכן נתונים. UPDATE הוא המשפט היחיד מבין השלושה בו קיימות הן גרסה חדשה והן גרסה ישנה של שורות, כך שאתה יכול להתייחס הן ל–deleted והן ל–inserted. ל–UPDATEs עם הפסוקית OUTPUT קיימים יישומים מעניינים רבים. אתן דוגמה לניהול תור הודעות או אירועים.

SQL Server 2005 מציג תשתית תורים חדשה לחלוטין ופלטפורמה הנקראת Service Broker המבוססת על תשתית זו.

מידע נוסף: לפרטים על תכנות עם Service Broker, אנא פנה לספר תכנות ב-T-SQL.

ניתן להשתמש ב– Service Broker כדי לפתח יישומים המנהלים תורים במסד הנתונים שלך. עם זאת, כאשר עליך לנהל תורים בקנה מידה קטן הרבה יותר, מבלי לערב את תשתית ופלטפורמת התורים החדשות, תוכל לעשות זאת על ידי שימוש בפסוקית ה–OUTPUT החדשה. כדי להדגים ניהול תור, הרץ את הקוד הבא, היוצר את טבלת Messages:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Messages') IS NOT NULL
  DROP TABLE dbo.Messages;
GO

CREATE TABLE dbo.Messages
(
  msgid   INT          NOT NULL IDENTITY ,
  msgdate DATETIME     NOT NULL DEFAULT(GETDATE()),
  msg     VARCHAR(MAX) NOT NULL,
  status  VARCHAR(20)  NOT NULL DEFAULT('new'),
  CONSTRAINT PK_Messages
    PRIMARY KEY NONCLUSTERED(msgid),
  CONSTRAINT UNQ_Messages_status_msgid
    UNIQUE CLUSTERED(status, msgid),
  CONSTRAINT CHK_Messages_status
    CHECK (status IN('new', 'open', 'done'))
);
```

לכל הודעה אתה מאחסן קוד הודעה, תאריך רישום, מלל הודעה, וסטטוס המציין האם ההודעה עדיין לא עובדה ('new'), מעובדת כעת ('open') או עברה כבר עיבוד ('done').

הקוד הבא מדמה session המייצר הודעות על ידי שימוש בלולאה המכניסה הודעה עם מלל רנדומאלי כל שנייה. הסטטוס של הודעות חדשות שנכנסות הוא 'new' מכיוון שלטור הסטטוס הוקצה ערך ברירת המחדל 'new'. הרץ קוד זה ממספר sessions בו זמנית:

```
SET NOCOUNT ON;
USE tempdb;
GO
DECLARE @msg AS VARCHAR(MAX);
WHILE 1=1
BEGIN
  SET @msg = 'msg' + RIGHT('000000000'
    + CAST(CAST(RAND()*2000000000 AS INT)+1 AS VARCHAR(10)), 10);
  INSERT INTO dbo.Messages(msg) VALUES(@msg);
  WAITFOR DELAY '00:00:01';
END
```

כמובן, באפשרותך לשחק עם תקופת ההשהיה כרצונך.

הקוד הבא מדמה session המעבד הודעות על ידי שימוש בצעדים הבאים:

1. צור לולאה אינסופית המעבדת הודעות בצורה תמידית.
2. נעל @n הודעות חדשות זמינות על ידי שימוש במשפט UPDATE TOP(@n) עם ה– READPAST hint כדי לדלג על שורות נעולות, ושנה את הסטטוס שלהן ל–'open'. @n מייצג קלט ניתן–להגדרה הקובע את המספר המקסימלי של הודעות שיעובדו בכל איטרציה.
3. אחסן את טורי ההודעות במשתנה הטבלה @Msgs על ידי שימוש בפסוקית OUTPUT.
4. עבד את ההודעות.
5. שנה את סטטוס ההודעות ל–'done' על ידי מיזוג טבלת Messages ומשתנה הטבלה @Msgs.
6. אם לא נמצאה כל הודעה חדשה בטבלת Messages, המתן שנייה אחת.

```
SET NOCOUNT ON;
USE tempdb;
GO

DECLARE @Msgs TABLE(msgid INT, msgdate DATETIME, msg VARCHAR(MAX));
DECLARE @n AS INT;
SET @n = 3;

WHILE 1 = 1
```

```
BEGIN
  UPDATE TOP(@n) dbo.Messages WITH(READPAST) SET status = 'open'
    OUTPUT inserted.msgid, inserted.msgdate, inserted.msg INTO @Msgs
    OUTPUT inserted.msgid, inserted.msgdate, inserted.msg
  WHERE status = 'new';

  IF @@rowcount > 0
  BEGIN
    PRINT 'Processing messages...';
    /* ...process messages here... */

    WITH UPD_CTE AS
    (
      SELECT M.status
      FROM dbo.Messages AS M
        JOIN @Msgs AS N
          ON M.msgid = N.msgid
    )
    UPDATE UPD_CTE
      SET status = 'done';

    DELETE FROM @Msgs;
  END
  ELSE
  BEGIN
    PRINT 'No messages to process.';
    WAITFOR DELAY '00:00:01';
  END
END
```

ניתן להריץ קוד זה ממספר sessions בו זמנית. תוכל להגדיל את מספר ה–sessions שיריצו קוד זה בהתבסס על קצב העיבוד אותו עליך לספק.

שים לב שרק למטרות הדגמה, כללתי במשפט ה–UPDATE הראשון פסוקית OUTPUT שנייה, המחזירה את ההודעות חזרה לשולח. אני מוצא שמשפט ה–UPDATE הזה יפה במיוחד מכיוון שהוא מקיף ארבעה חידושי T-SQL שונים ב– SQL Server 2005: UPDATE TOP, TOP עם ביטוי קלט, הפסוקית OUTPUT וה– READPAST hint במשפטי שינוי נתונים. ה– READPAST hint היה זמין ב– SQL Server 2000, אך רק עבור שאילתות SELECT.

משפטי SELECT ו–UPDATE של הצבה

סעיף זה מכסה משפטים המקצים ערכים למשתנים ובמקרה של UPDATE, יכולים בו בזמן לשנות נתונים. קיימות מספר סוגיות ערמומיות עם הקצאות כאלה שכדאי שתהיה מודע אליהן. היכרות עם הדרך בה עובדות הקצאות ב–T-SQL חשובה לצורך תכנות נכון – כלומר, כדי לתכנת את מה שהתכוונת.

SELECT של הצבה

אתחיל במשפטי SELECT של הצבה. T-SQL תומך בהצבת ערכים למשתנים על ידי שימוש במשפט SELECT, אך צורת ה–ANSI של הצבה שגם היא נתמכת על ידי T-SQL, היא להשתמש במשפט SET. אם כך, ככלל, אלא אם כן יש סיבה טובה במיוחד לנהוג אחרת, זהו נוהג טוב להיצמד לשימוש ב–SET. אתאר מקרים בהם ייתכן שתרצה להשתמש ב–SELECT מכיוון שיש לו יתרונות על פני SET במקרים אלו. עם זאת, כפי שאציג מייד, עליך להיות מודע לכך שכאשר אתה משתמש ב–SELECT, הקוד שלך מועד יותר לטעויות.

כדוגמה לדרך בה עובד משפט SELECT של הצבה, נניח שעליך להקצות למשתנה @EmpID את קוד העובד ששם המשפחה שלו זהה לתבנית מסוימת (@pattern). אתה מניח שרק עובד אחד יתאים לתבנית שלך. הקוד הבא, המשתמש ב–SELECT של הצבה, אינו עונה על הדרישות:

```
USE Northwind;

DECLARE @EmpID AS INT, @Pattern AS NVARCHAR(100);

SET @Pattern = N'Davolio'; -- Try also N'Ben-Gan', N'D%';
SET @EmpID = 999;

SELECT @EmpID = EmployeeID
FROM dbo.Employees
WHERE LastName LIKE @Pattern;

SELECT @EmpID;
```

בהינתן 'N'Davolio כתבנית הקלט, אתה מקבל את קוד העובד 1 במשתנה @EmpID. במקרה זה, רק עובד אחד התאים למסנן. עם זאת, אם אתה מקבל תבנית שאינה מתאימה לשום שם משפחה קיים בטבלת Employees (למשל, 'N'Ben-Gan), ההצבה אינה מתרחשת אפילו פעם אחת. התוכן של המשתנה @EmpID נשאר כפי שהיה לפני ההצבה – 999 (ערך זה משמש למטרות הדגמה). אם ניתנה לך תבנית המתאימה ליותר מאשר שם משפחה אחד (למשל, 'N'D%), קוד זה יבצע מספר הצבות, הדורסות את הערך הקודם

ב–@EmpID בכל הצבה. הערך הסופי של @EmpID, יהיה קוד העובד מהשורה המתאימה אליה SQL Server ניגש במקרה אחרונה.

דרך בטוחה הרבה יותר להצבת קוד העובד המתאים למשתנה @EmpID היא להשתמש במשפט SET כלהלן:

```
DECLARE @EmpID AS INT, @Pattern AS NVARCHAR(100);

SET @Pattern = N'Davolio'; -- Try also N'Ben-Gan', N'D%';
SET @EmpID = 999;

SET @EmpID = (SELECT EmployeeID
              FROM dbo.Employees
              WHERE LastName LIKE @Pattern);

SELECT @EmpID;
```

אם רק עובד אחד מתאים, תקבל את קוד העובד במשתנה @EmpID. אם שום עובד אינו מתאים, תת–השאילתה תשנה את @EmpID ל–NULL. כאשר אתה מקבל NULL, אתה יודע שלא היו כל התאמות. אם מספר עובדים מתאימים, תקבל הודעת שגיאה האומרת שתת–השאילתה החזירה יותר מערך אחד. במקרה כזה, תבין שיש בעיה בהנחות שלך או בעיצוב של הקוד שלך, אך הבעיה תצוף במקום שתישאר חבויה.

כאשר אתה מבין כיצד SELECT של הצבה עובד, תוכל להשתמש ביתרונות שבו. לדוגמה, משפט SET יכול להציב ערך רק במשתנה אחד בכל זמן. SELECT של הצבה יכול להקצות ערכים למספר משתנים בתוך אותו משפט.

עם קוד מעוצב–היטב, יכולת זו יכולה לתת לך יתרונות ביצועים. לדוגמה, הקוד הבא מציב למשתנים את השם הפרטי ושם המשפחה של עובד נתון:

```
DECLARE @FirstName AS NVARCHAR(10), @LastName AS NVARCHAR(20);

SELECT @FirstName = NULL, @LastName = NULL;

SELECT @FirstName = FirstName, @LastName = LastName
FROM dbo.Employees
WHERE EmployeeID = 3;

SELECT @FirstName, @LastName;
```

שים לב שקוד זה משתמש במפתח הראשי כדי לסנן עובד, כלומר שאינך יכול לקבל יותר משורה אחת חזרה. הקוד גם מאתחל את המשתנים @FirstName ו–@LastName עם NULLs. אם שום עובד לא מתאים, המשתנים פשוט יישארו עם NULLs. סוג זה של הצבה

שימושי במיוחד בטריגרים כאשר ברצונך לקרוא טורים מהטבלאות המיוחדות inserted ו–deleted לתוך המשתנים שלך, לאחר שאתה מוודא ששורה אחת בלבד הושפעה.

טכנית, תוכל להישען על העובדה ש–SELECT של הצבה מבצע מספר הצבות, כאשר קיימות מספר שורות מתאימות. לדוגמה, תוכל לבצע חישובי צבירה, כגון שרשור כל קודי ההזמנה ללקוח נתון:

```
DECLARE @Orders AS VARCHAR(8000), @CustomerID AS NCHAR(5);
SET @CustomerID = N'ALFKI';
SET @Orders = '';

SELECT @Orders = @Orders + CAST(OrderID AS VARCHAR(10)) + ';'
FROM dbo.Orders
WHERE CustomerID = @CustomerID;

SELECT @Orders;
```

עם זאת, קוד זה אינו נתמך על ידי התקן, ואינך יכול להבטיח את סדר ההצבה. ראיתי ניסיונות שעשו תוכניתנים לשלוט על סדר ההצבה על ידי הוספת פסוקית ORDER BY כלהלן:

```
DECLARE @Orders AS VARCHAR(8000), @CustomerID AS NCHAR(5);
SET @CustomerID = N'ALFKI';
SET @Orders = '';

SELECT @Orders = @Orders + CAST(OrderID AS VARCHAR(10)) + ';'
FROM dbo.Orders
WHERE CustomerID = @CustomerID
ORDER BY OrderDate, OrderID;

SELECT @Orders;
```

אך אינך יכול להכריח את ה–optimizer למיין בטרם ביצוע ההצבה. אם ה–optimizer בוחר למיין לאחר ההצבה, לפסוקית ה– ORDER BY כאן לא תהיה ההשפעה הרצויה. אתה מבין שכאשר אתה מציין פסוקית ORDER BY אתה עשוי לקבל מצב ביניים של המשתנה. בקיצור, עדיף לא להישען על שיטות כאלה. קיימות מספיק שיטות נתמכות ומובטחות עבור חישובים כאלה מהן ניתן לבחור, רבות מהן כיסיתי בפרק 6.

UPDATE של הצבה

T-SQL תומך גם בתחביר UPDATE שאינו בתקן, שיכול להציב ערכים במשתנים בנוסף לשינוי נתונים. כדי להדגים את השיטה, ראשית הרץ את הקוד הבא, היוצר את טבלה T1 וממלא אותה בנתונים. לדוגמה:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  col1 INT        NOT NULL,
  col2 VARCHAR(5) NOT NULL
);

INSERT INTO dbo.T1(col1, col2) VALUES(0, 'A');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'B');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'B');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'A');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'A');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
INSERT INTO dbo.T1(col1, col2) VALUES(0, 'C');
```

כרגע, לטבלה T1 אין מפתח ראשי ואין כל דרך לזהות ייחודית את השורות. נניח שהיית רוצה להקצות מספרים שלמים ייחודיים לטור col1 ואז להפוך אותו למפתח הראשי. תוכל להשתמש ב–UPDATE של הצבה כדי לבצע משימה זו:

```
DECLARE @i AS INT;
SET @i = 0;
UPDATE dbo.T1 SET @i = col1 = @i + 1;
```

קוד זה מכריז על המשתנה @i ומאתחל אותו עם 0. אז משפט ה–UPDATE סורק את הנתונים ולכל שורה משנה את הערך הנוכחי של col1 לערך @i + 1, ואז משנה את הערך של המשתנה @i לערך החדש של col1. לוגית, הפסוקית SET מקבילה ל– SET col1 = @i + 1, @i = @i + 1. עם זאת, במשפט UPDATE כזה, אין לך כל דרך לשלוט על הסדר לפיו ייסרקו ויעודכנו השורות ב–T1. לדוגמה, טבלה 4–8 מציגה כיצד עשוי להיראות התוכן של T1 לאחר ההקצאה.

טבלה 8–4: תוכן של T1 לאחר הפעלת UPDATE של הצבה

col1	*col2*
1	A
2	B
3	C
4	C
5	C
6	B
7	A
8	A
9	C
10	C

אך זכור שהתוכן עשוי להיות שונה. כל עוד לא חשוב לך הסדר לפיו הנתונים נסרקים ומעודכנים, אתה עשוי להיות מרוצה משיטה זו. היא מהירה מאוד, שכן היא סורקת את הנתונים פעם אחת בלבד.

SQL Server 2005 מאפשר לך למלא את המשימה בדרך אלגנטית, כאשר אתה יכול להבטיח שמספרי השורה של התוצאה יתבססו על סדר מבוקש. כדי לעשות זאת, צור UPDATE על CTE המחשב מספרי שורה בהתבסס על כל סדר רצוי:

```
WITH T1RN AS
(
  SELECT col1, ROW_NUMBER() OVER(ORDER BY col2) AS RowNum
  FROM dbo.T1
)
UPDATE T1RN SET col1 = RowNum;
```

טבלה 8–5 מציגה את התוכן של T1 לאחר ה–UPDATE.

טבלה 8–5: תוכן של T1 לאחר הפעלת UPDATE מול CTE

col1	*col2*
1	A
4	B
6	C

col1	*col2*
7	C
8	C
5	B
2	A
3	A
9	C
10	C

בשלב זה, אתה בטח כבר מבין מדוע החידושים האהובים עלי ב– SQL Server 2005 הם הפונקציה ROW_NUMBER ו–CTEs.

שיקולי ביצועים אחרים

בסעיף זה, אספק רקע קצר על עיצוב וארכיטקטורה של מערכות מרובות–שינויים. חשבתי שרקע כזה עשוי להיות מעניין עבור תוכניתנים, במיוחד כדי להבין שישנם גורמים רבים המעורבים בעיצוב מערכות, כאשר המטרה היא להשיג ביצועים טובים בעת ביצוע שינויי נתונים. אם המונחים המתוארים בסעיף זה אינם מוכרים לך, תוכל לדלג עליו.

מידע נוסף: תוכל למצוא התייחסות מעמיקה יותר של הנושא והסברים מפורטים יותר על המונחים הנידונים כאן בספר Inside Microsoft SQL Server 2005 – The Storage Engine (Microsoft Press, 2006) מאת Kalen Delaney.

כאשר אתה מעצב מערכות מרובות–שינויים, עליך לקחת בחשבון מספר דברים חשובים. אלא אם כן אני מציין מפורשות אחרת, הדיון כאן מתאים לכל סוג של שינוי נתונים (הוספה, מחיקה ועדכון נתונים).

במונחים של עיצוב פיסי של מערכת מסד הנתונים שלך, זכור ששינויים משפיעים על שני חלקים של מסד הנתונים: הנתונים ולוג הטרנזקציות. כאשר מתרחש שינוי נתונים, SQL Server מחפש תחילה אחר הדפים בהם צריך להתבצע שינוי ב–cache. אם דפי היעד נמצאים כבר ב–cache, SQL Server מבצע את השינויים בהם שם. אם הם אינם ב–cache, SQL Server יטען תחילה את דפי היעד מחלק הנתונים של מסד הנתונים לתוך ה–cache, ואז יבצע בהם שינויים שם. SQL Server כותב את השינוי בלוג הטרנזקציות (אדון מייד בהיבטים של כתיבה ללוג). SQL Server מריץ מדי פעם תהליך checkpoint, המוריד דפים מלוכלכים (שעברו שינוי) מה–cache לחלק הנתונים של מסד הנתונים על הדיסק. עם זאת, SQL Server יוריד רק דפים מלוכלכים עבורם השינוי נכתב כבר בלוג הטרנזקציות.

לוג הטרנזקציות בעיקרון מספק את היבט הקיימות (durability) של טרנזקציות (ה–D בהיבטי ACID של טרנזקציות), מה שמאפשר יכולות rollback ו– roll-forward.

פעילויות כתיבה מרובות לחלק הנתונים של מסד הנתונים יכולות להיעשות בצורה מקבילה על ידי שימוש במספר דיסקים. לפיכך, פרישת הנתונים על פני מספר דיסקים הוא גורם מפתח מבחינת ביצועים. כמה שיותר דיסקים, יותר טוב. מערכת RAID 10 היא לרוב הסוג האופטימלי של מערכות RAID עבור סביבות בהן מתבצעת כתיבה מרובה. מערכת RAID 10 מבצעת הן פרישה של הנתונים והן mirror לכל דיסק. היא אינה מערבת כל חישובי parity. החיסרון העיקרי שלה הוא שהיא יקרה, מכיוון שמחצית מהדיסקים במערך משמשים ל–mirror של המחצית השנייה. מערכת RAID 5 היא לרוב בחירה לא מוצלחת למערכות בהן מתבצעת כתיבה מרובה מכיוון שהיא מערבת חישובי parity בכל כתיבה. היתרון העיקרי שלה הוא העלות הנמוכה, מכיוון שעבור מערך של n דיסקים, רק 1/n משמשים ל–parity. עם זאת, במקרים רבים מערכת RAID 10 יכולה להביא לשיפורי ביצועים, עבור פעולות כתיבה של למעלה מ–50 אחוז מאשר מערכת RAID 5, כך שהעלות הגבוהה שלה היא לרוב כדאית. עבור מערכות אשר בעיקר קוראות נתונים (לדוגמה, data warehouses), מערכת RAID 5 מספיקה עבור חלק הנתונים כל עוד לתהליך החילוץ, המרה וטעינה – ETL (Extract, Transform and Load) יש חלון זמן מספיק (לרוב במהלך הלילה). בעוד שבמהלך היום, יישומים רק קוראים נתונים מה– data warehouse.

באשר ללוג הטרנזקציות, הארכיטקטורה שלו היא כזו שהוא יכול להיכתב אך ורק בצורה סינכרונית – כלומר, רק באופן רציף. לפיכך, לפרישת הלוג על פני מספר דיסקים אין כל יתרון אלא אם כן יש לך גם תהליכים הקוראים מהלוג. דוגמה אחת לתהליך הקורא מלוג הטרנזקציות היא רפליקציה של לוג הטרנזקציות (transaction log replication). דוגמה אחרת היא גישה לטבלאות deleted ו–inserted בטריגרים ב– SQL Server 2000.

טבלאות אלו ב– SQL Server 2000 משקפות את חלק הלוג המכיל את השינוי שהפעיל את הטריגר. ב– SQL Server 2005, שורות deleted ו–inserted מבוססות על טכנולוגיית ה–row-versioning החדשה, המתחזקת גרסאות של שורות במסד הנתונים tempdb. המשמעות היא שכאשר אתה ניגש ל–inserted ו–deleted, SQL Server 2005 יסרוק נתונים ב–tempdb ולא בלוג הטרנזקציות. בכל אופן, זהו נוהג טוב להפריד את לוג הטרנזקציות לדיסק משל עצמו כדי להימנע מהפרעה לפעילות שלו. כל הפרעה לפעילות של לוג הטרנזקציות מעכבת בסופו של דבר את הכתיבה לחלק הנתונים של מסד הנתונים. אלא אם כן יש לך תהליכים הקוראים מלוג הטרנזקציות, מערכת RAID 1 מספיקה. מערכת RAID 1 מבצעת רק mirror לדיסק ואינה פורשת את הנתונים. אך אם יש לך גם תהליכים אינטנסיביים הקוראים מהלוג, מערכת RAID 10 תהיה בחירה טובה יותר.

באשר ל–tempdb, תמיד טוב לפרוש אותו על ידי שימוש במערכת RAID 10. דבר זה נכון הן לעיבוד טרנזקציות online (OLTP) והן למערכות data warehouse, מכיוון ש– SQL Server שומר נתונים ב–tempdb עבור פעילויות רקע רבות הקשורות הן לקריאות והן לכתיבות. כוונון של tempdb הופך להיות אפילו חשוב יותר ב– SQL Server 2005, מכיוון שטכנולוגיית ה–row-versioning החדשה המשמשת מספר פעילויות, מאחסנת וקוראת

גרסאות שורה מ–tempdb. פעילויות המשתמשות ב–row-versioning כוללות פעולות אינדקס online, בניית הטבלאות deleted ו–inserted בטריגרים, snapshot isolations החדשות ו– MARS (Multiple Active Result Sets).

לאופן הסינכרוני בו SQL Server כותב ללוג הטרנזקציות יש השפעה משמעותית על שינויי נתונים. הביצועים של שינויים הנכתבים ללוג בצורה מלאה, לעיתים קרובות מוגבלים על ידי הזמן שלוקח לכתוב ללוג. יתרה מכך, ברגע שקצב הכתיבות יגיע להספק של הדיסק עליו יושב הלוג, שינויי נתונים יתחילו לחכות לכתיבות ללוג. כאשר לוג הטרנזקציות הופך לצוואר הבקבוק, עליך לשקול פיצול של מסד הנתונים למספר מסדים, כל אחד עם לוג טרנזקציות משל עצמו היושב על דיסק נפרד.

עם רקע זה, תוכל לראות שבמקרים רבים הביצועים של שינויי נתונים באופן כללי – ובמיוחד הוספת נתונים – יהיו תלויים במידה רבה בכמות הכתיבה ללוג. כאשר מעצבים תהליכי הוספת נתונים, אחד מהשיקולים העיקריים צריך להיות כמות הכתיבה ללוג.

SQL server יבצע תמיד כתיבה מלאה ללוג של שינויי נתונים אלא אם כן מתקיימים שני תנאים: מודל ההתאוששות של מסד הנתונים אינו FULL, והפעולה נחשבת לפעולת BULK. פעולת BULK המופעלת על מסד נתונים עם מודל נתונים שאינו FULL נכתבת ללוג בצורה מינימלית. כתיבה מינימלית ללוג משמעותה רישום רק של כתובות ה–extents שהוקצו במהלך הפעולה כדי לתמוך ביכולות rollback, בניגוד לכתיבה ללוג של השינוי כולו. פעולות BULK כוללות יצירה או בנייה–מחדש של אינדקס, הוספות נתונים המשתמשות במנוע ה–BULK (לדוגמה, BULK INSERT), SELECT INTO, מניפולציה על LOB (Large Objects). דוגמאות למניפולציה של LOB ב– SQL Server 2000 כוללות WRITETEXT ו– UPDATETEXT. SQL Server 2005 תומך גם בשיטות מניפולציה של LOB חדשות, ביניהן שימוש ב– BULK rowset provider והשיטה WRITE. שינוי נתונים הנכתב ללוג בצורה מינימלית, הוא לרוב מהיר יותר משמעותית מאשר שינוי הנכתב ללוג בצורה מלאה.

אם כך, כאשר מעצבים תהליכי הוספת נתונים, האפשרות הראשונה צריכה להיות שימוש בפעולות BULK. האפשרות השנייה צריכה להיות שימוש ב–INSERT מבוסס–סטים של שורות מרובות (INSERT SELECT), והאפשרות האחרונה צריכה להיות INSERTs בודדים. הכלל הבסיסי הוא כמות הכתיבה ללוג. INSERTs בודדים יוצרים כמות רבה יותר משמעותית של כתיבה ללוג, מאשר INSERTs של שורות מרובות, אשר בתורם יוצרים כמות רבה יותר של כתיבה ללוג מאשר פעולות BULK הרצות במסד נתונים עם מודל התאוששות שאינו FULL.

גורם נוסף שישפיע על הביצועים של הוספות הנתונים שלך הוא גודל הטרנזקציה. זהו נושא ערמומי.

התרחיש הגרוע ביותר הוא הוספת שורות בודדות, כל אחת בטרנזקציה משל עצמה. במקרה כזה, לא זו בלבד שה–INSERT נרשם ללוג בצורה מלאה, כל INSERT גורם לכתיבה של שלוש רשומות ללוג הטרנזקציות: BEGIN TRAN, ה–INSERT עצמו

ו– COMMIT TRAN. מלבד הכתיבה המוגברת ללוג, עליך לשקול את התקורה המעורבת בתחזוקה של טרנזקציה כזו (לדוגמה, ביצוע נעילות, שחרור נעילות וכך הלאה). כדי לשפר ביצועים של הוספת נתונים, תרצה לעטוף כל מקטע של שורות מרובות בתוך טרנזקציה בודדת.

השאלה היא, מהו הגודל האופטימלי של טרנזקציה? זהו החלק הקשה. העניינים אינם שחור ולבן, כך שטרנזקציות של שורה בודדת יהיו התרחיש הגרוע ביותר וטרנזקציה אחת גדולה תהיה התרחיש הטוב ביותר. בפועל, ככל שהטרנזקציה גדלה, ביצועי הוספת נתונים משתפרים עד לנקודה כלשהי. הנקודה בה ביצועי הוספת נתונים מתחילים להתדרדר, היא כאשר תחזוקת הטרנזקציה הענקית מערבת תקורה רבה מדי ל– SQL Server. גורמים רבים משפיעים על גודל הטרנזקציה שתיתן לך את ביצועי הוספת הנתונים האופטימלים. אלו כוללים את החומרה, מבנה מסד הנתונים שלך (נתונים, לוג), עיצוב אינדקסים, וכך הלאה. אם כך, במונחים מעשיים, הדרך הטובה ביותר לגילוי גודל הטרנזקציה האופטימלי עבור טבלה מסוימת היא להשתמש בבוחן–ביצועים. פשוט בחן את תהליך הוספת הנתונים שלך עם גדלי טרנזקציות שונים, תוך שאתה מגדיל או מקטין את מספר השורות שאתה עוטף בטרנזקציה יחידה בהתבסס על הביצועים שאתה מקבל. לאחר שתכוונן את גודל הטרנזקציה מספר פעמים בצורה כזו, תמצא את הגודל האופטימלי.

כאשר מכוונגים הוספת נתונים בשורות בודדות, בסביבת הבדיקות שלך אתה יכול ליצור לולאה המוסיפה שורה בכל איטרציה, ולתחזק מונה בלולאת ה–INSERT שלך. פתח טרנזקציה חדשה לפני כל n איטרציות, כאשר n הוא מספר השורות שאתה מעוניין לבדוק את הוספתו בטרנזקציה יחידה, ובצע commit לטרנזקציה לאחר כל n איטרציות. ראיתי סביבות תפעוליות אשר אוספות את הנתונים להוספה מסביבות המקור, ויוצרות לולאה כזו כדי להוסיף את הנתונים לטבלאות היעד. כמובן, לא כל מערכת מעוצבת לאפשר לוגיקת לולאות כזו.

תהליך זה (שליטה בגודל הטרנזקציה של הוספת נתונים) רלוונטי לכל סוג של הוספה – כלומר, INSERTs של שורות בודדות, INSERTs מבוססי–סטים של שורות מרובות (INSERT SELECT) ו– bulk inserts. ב– bulk inserts (לדוגמה, שימוש בפקודה BULK INSERT), כברירת מחדל, כל ההוספה נחשבת לטרנזקציה יחידה, אך באפשרותך לשלוט במספר השורות בכל טרנזקציה ובמידת האופטימליות של הוספת הנתונים, על ידי קביעת האפשרויות BATCHSIZE ו– ROWS_PER_BATCH.

סיכום

שינויי נתונים מערבים אתגרים רבים. עליך להכיר את הארכיטקטורה וה–internals של SQL Server כדי לעצב מערכות שיכולות להתמודד עם נפחי נתונים גדולים, ושינויי נתונים בקנה מידה גדול. קיימות גם בעיות לוגיות מאתגרות רבות הקשורות בשינויי נתונים, כגון תחזוקת רצף שהגדרת בעצמך, מחיקת שורות עם נתונים כפולים והקצאת ערכים ייחודיים לשורות קיימות. בפרק זה, כיסיתי היבטי ביצועים של שינויי נתונים כמו גם היבטים לוגיים, וכן הצגתי מספר לא קטן של שיטות מפתח שימושיות.

9

גרפים, עצים, היררכיות ושאילתות רקורסיביות

פרק זה עוסק בטיפול במבני נתונים מיוחדים הנקראים גרפים, עצים והיררכיות ב– Microsoft SQL Server, תוך שימוש ב–T-SQL. מבין השלושה, המונח השגור ביותר אולי בין תוכניתני T-SQL הוא **היררכיה**, ומונח זה משמש לעיתים אפילו כאשר מבנה הנתונים המדובר אינו באמת היררכיה. אתחיל בסעיף **טרמינולוגיה** שיתאר כל מבנה נתונים ויבהיר את פני הדברים.

טיפול (הצגה, תחזוקה ומניפולציה) בגרפים, עצים והיררכיות ב–RDBMS רחוק מלהיות טריוויאלי. אדון בשתי גישות מרכזיות, אחת המתבססת על לוגיקה איטרטיבית/רקורסיבית, והאחרת מתבססת על קיום מידע נוסף במסד הנתונים המתאר את מבנה הנתונים.

מעניין שעל אף שמבני נתונים אלה יושמו ועדיין מיושמים בצורה נרחבת במערכות ניהול מסדי נתונים רלציוניים (RDBMSs), תמיכה בשאילתות רקורסיביות הוצגה רק בתקן ANSI SQL:1999. SQL Server 2005 אימץ בפעם הראשונה במידה מסוימת את ההרחבות לשאילתות רקורסיביות של ANSI SQL:1999 ב–T-SQL.

בפרק זה אדון בפתרונות המשתמשים בשאילתות הרקורסיביות החדשות ב– SQL Server 2005, כמו גם בפתרונות הניתנים ליישום בגרסאות קודמות של SQL Server.

טיפ: אני מעודד אתכם גם לקרוא על מודל האינטרוולים של וואדים טרופשקו (Vadim Tropashko) בכתובת www.dbazine.com. זהו מודל יפהפה, מאוד מעניין אינטלקטואלית. וואדים דן בסוגיות מעשיות כמו הטמעה וביצועים. עם זאת, אני מוצא שהמודל של וואדים מורכב מדי לתפיסה כוללת עבור מרבית בני–התמותה (כולל אותי), כך שלא אכלול אותו כאן. הפתרונות שיופיעו כאן, מצד שני, יהיו פשוטים יחסית להבנה וליישום על ידי תוכניתני T–SQL מנוסים. בטרם תנסה לקרוא את החומר של וואדים, ודא שיש לך מספיק קפה ומספיק שעות שינה.

כפי שהבטחתי אתחיל בסעיף **טרמינולוגיה** המתאר גרפים, עצים והיררכיות.

טרמינולוגיה

שים לב: ההסברים בסעיף זה מבוססים על הגדרות של NIST (National Institute of Standards and Technology). ביצעתי מספר שינויים והוספתי מעט מלל להגדרות המקוריות, כדי להפוך אותן לפחות פורמליות וכדי לשמור על רלוונטיות לנושא המדובר (T–SQL).

להגדרות שלמות ופורמליות יותר של גרפים, עצים ומונחים קשורים, אנא פנה לכתובת: http://www.nist.gov/dads.

גרפים

גרף הוא סט של פרטים המחוברים על ידי **מקצועות** (edges). כל פרט נקרא **צומת** (vertex או node). **מקצוע** הוא חיבור בין שני צמתים של גרף.

גרף הוא מונח כולל עבור מבנה נתונים, ותרחישים רבים יכולים להיות מיוצגים כגרפים – למשל, מבנה ארגוני, עץ מוצר (BOM), מערכת כבישים וכך הלאה. כדי לצמצם את סוג הגרף למקרה ספציפי יותר, עליך לזהות את תכונותיו:

- **מכוון/בלתי–מכוון (Directed/Undirected)** בגרף מכוון (ידוע גם כ–digraph), קיים כיוון או סדר בין שתי הצמתים של מקצוע. לדוגמה, בגרף עץ מוצר למוצרי בית–קפה, לאטה מכיל חלב ולא הפוך. קיים מקצוע (יחסי הכלה) בגרף לזוג הצמתים/הפרטים (Latte, Milk), אך לא קיים מקצוע לזוג (Milk, Latte).

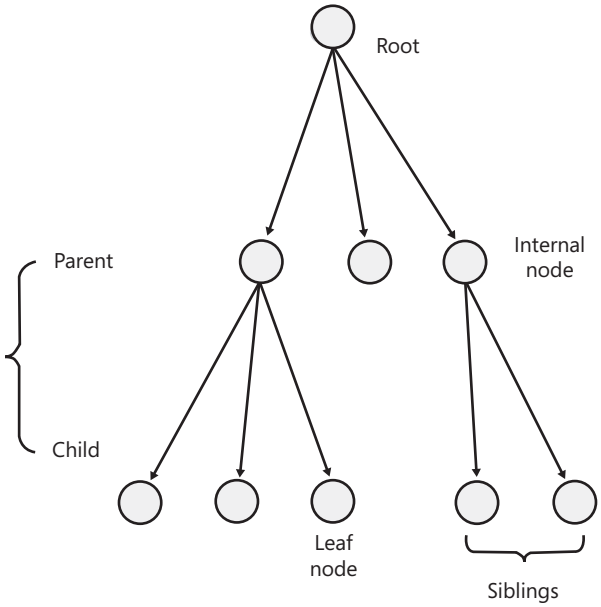
  בגרף **בלתי–מכוון**, כל מקצוע מחבר פשוט בין שני צמתים, ללא סדר מסוים. למשל, במערכת כבישים קיימת דרך בין לוס–אנג'לס לסן–פרנסיסקו. המקצוע (דרך) בין הצמתים (ערים) לוס–אנג'לס וסן–פרנסיסקו יכול להיות מבוטא באחת משתי צורות: {Los Angeles, San Francisco} או {San Francisco, Los Angeles}.
- **לא–מעגלי (Acyclic)** גרף לא–מעגלי הוא גרף שאין בו מחזוריות – כלומר, לא קיים **מסלול** המתחיל ומסתיים באותו צומת – למשל, מבנה ארגוני ועץ מוצר. גרף מכוון לא–מעגלי (directed acyclic graph) ידוע גם כ–DAG. אם קיימים מסלולים המתחילים ומסתיימים באותו צומת – כפי שלרוב יש במערכת כבישים – הגרף הוא מעגלי.
- **קשור (Connected)** גרף קשור הוא גרף בו קיים מסלול בין כל זוג צמתים – למשל, מבנה ארגוני.

עצים

עץ הוא מקרה מיוחד של גרף – גרף קשור לא–מעגלי.

הגישה לעץ מתחילה להתבצע בצומת **השורש** (root). כל צומת הוא או **עלה** (leaf) או **צומת פנימית** (internal node). לצומת פנימית קיים צומת **בן** (child) אחד או יותר והוא נקרא **ההורה** (parent) של צמתי הבנים שלו. כל הבנים של אותו צומת הם **אחים** (siblings). בניגוד למה שנראה בעץ פיסי, השורש לרוב מוצג בראש המבנה והעלים מוצגים בתחתית (ראה תרשים 1–9).

תרשים 1–9: עץ



יער הוא אוסף של עץ אחד או יותר – למשל, דיונים בפורומים יכולים להיות מיוצגים כיער כאשר כל דיון הוא עץ.

היררכיות

ישנם תרחישים היכולים להיות מוגדרים כהיררכיה, המעוצבים כגרף מכוון לא–מעגלי – למשל, הורשה בין טיפוסים/מחלקות בתכנות מוכוון–אובייקטים ויחסי עובד–מעביד במבנה ארגוני. בראשון, מקצועות הגרף מייצגים את ההורשה. מחלקות יכולות לרשת מתודות ומאפיינים ממחלקות אחרות (וייתכן גם ממספר מחלקות). בתרחיש השני, המקצועות מייצגים את יחסי העובד–מעביד בין עובדים. שים לב לטבע הלא–מעגלי, המכוון של תרחישים אלו. שרשרת הניהול והאחריות בחברה, למשל, אינה יכולה להסתובב במעגלים.

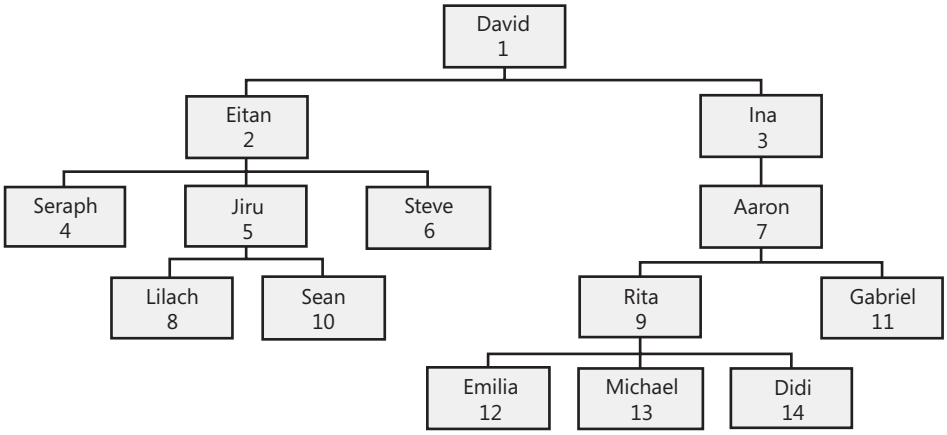
תרחישים

לאורך הפרק אשתמש בשלושה תרחישים: מבנה ארגוני (עץ, היררכיה), עץ מוצר (גרף לא–מעגלי מכוון – DAG), ומערכת כבישים (גרף מעגלי בלתי–מכוון). שים לב מה מבדיל בין עץ לבין DAG. כל העצים הם DAGs, אך לא כל ה–DAGs הם עצים. בעץ, לפרט יכול להיות הורה אחד לכל היותר; קיימות היררכיות ניהול, בהן לעובד עשוי להיות יותר ממנהל אחד.

מבנה ארגוני

המבנה הארגוני בו אשתמש מוצג גרפית בתרשים 2–9.

תרשים 2–9: מבנה ארגוני



כדי ליצור את טבלת Employees ולמלא אותה בנתונים לדוגמה, הרץ את הקוד בקטע–קוד 9–1. התוכן של טבלת Employees מוצג בטבלה 9–1.

קטע–קוד 9–1: מבנה ונתונים לדוגמה של טבלת Employees

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Employees') IS NOT NULL
  DROP TABLE dbo.Employees;
GO
```

```
CREATE TABLE dbo.Employees
(
  empid   INT         NOT NULL PRIMARY KEY,
  mgrid   INT         NULL     REFERENCES dbo.Employees,
  empname VARCHAR(25) NOT NULL,
  salary  MONEY       NOT NULL,
  CHECK (empid <> mgrid)
);

INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(1, NULL, 'David', $10000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(2, 1, 'Eitan', $7000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(3, 1, 'Ina', $7500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(4, 2, 'Seraph', $5000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(5, 2, 'Jiru', $5500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(6, 2, 'Steve', $4500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(7, 3, 'Aaron', $5000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(8, 5, 'Lilach', $3500.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(9, 7, 'Rita', $3000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(10, 5, 'Sean', $3000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(11, 7, 'Gabriel', $3000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(12, 9, 'Emilia' , $2000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(13, 9, 'Michael', $2000.00);
INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(14, 9, 'Didi', $1500.00);

CREATE UNIQUE INDEX idx_unc_mgrid_empid ON dbo.Employees(mgrid, empid);
```

טבלה 9–1: תוכן טבלת Employees

empid	*mgrid*	*empname*	*salary*
1	NULL	David	10000.0000
2	1	Eitan	7000.0000
3	1	Ina	7500.0000
4	2	Seraph	5000.0000
5	2	Jiru	5500.0000
6	2	Steve	4500.0000
7	3	Aaron	5000.0000
8	5	Lilach	3500.0000
9	7	Rita	3000.0000
10	5	Sean	3000.0000
11	7	Gabriel	3000.0000
12	9	Emilia	2000.0000
13	9	Michael	2000.0000
14	9	Didi	1500.0000

טבלת Employees מייצגת היררכיית ניהול כרשימת–סמיכות (adjacency list), כאשר המנהל והעובד מייצגים את צמתי ההורה והבן, בהתאמה.

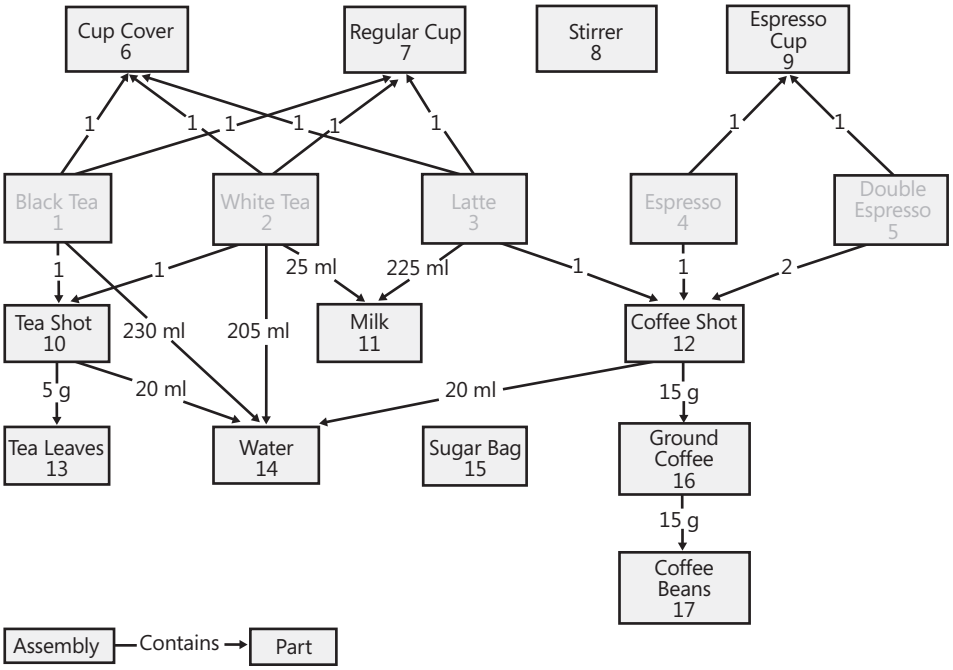
עץ מוצר (BOM)

אשתמש בעץ מוצר BOM (Bill of Materials) של מוצרי בית–קפה, המוצג גרפית בתרשים 9–3.

כדי ליצור את הטבלאות Parts ו–BOM ולמלא אותן בנתונים לדוגמה, הרץ את הקוד בקטע–קוד 9–2. התוכן של טבלאות Parts ו–BOM מוצג בטבלאות 9–2 ו–9–3.

שים לב שהתרחיש הראשון (מבנה ארגוני) דורש טבלה אחת בלבד, מכיוון שהוא מעוצב כעץ; הן מקצוע (manager, employee) והן צומת (employee) יכולים להיות מיוצגים על ידי אותה שורה. תרחיש עץ המוצר דורש שתי טבלאות, מכיוון שהוא מעוצב כ–DAG, כאשר לכל צומת עשויים להוביל מספר מסלולים; מקצוע (assembly, part) מיוצג על ידי שורה בטבלה BOM, וצומת (part) מיוצג על ידי שורה בטבלת Parts.

תרשים 3–9: עץ מוצר



קטע–קוד 2–9: מבנה ונתונים לדוגמה של טבלאות Parts ו–BOM

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.BOM') IS NOT NULL
  DROP TABLE dbo.BOM;
GO
IF OBJECT_ID('dbo.Parts') IS NOT NULL
  DROP TABLE dbo.Parts;
GO
CREATE TABLE dbo.Parts
(
  partid   INT         NOT NULL PRIMARY KEY,
  partname VARCHAR(25) NOT NULL
);

INSERT INTO dbo.Parts(partid, partname) VALUES( 1, 'Black Tea');
INSERT INTO dbo.Parts(partid, partname) VALUES( 2, 'White Tea');
INSERT INTO dbo.Parts(partid, partname) VALUES( 3, 'Latte');
```

```
INSERT INTO dbo.Parts(partid, partname) VALUES( 4, 'Espresso');
INSERT INTO dbo.Parts(partid, partname) VALUES( 5, 'Double Espresso');
INSERT INTO dbo.Parts(partid, partname) VALUES( 6, 'Cup Cover');
INSERT INTO dbo.Parts(partid, partname) VALUES( 7, 'Regular Cup');
INSERT INTO dbo.Parts(partid, partname) VALUES( 8, 'Stirrer');
INSERT INTO dbo.Parts(partid, partname) VALUES( 9, 'Espresso Cup');
INSERT INTO dbo.Parts(partid, partname) VALUES(10, 'Tea Shot');
INSERT INTO dbo.Parts(partid, partname) VALUES(11, 'Milk');
INSERT INTO dbo.Parts(partid, partname) VALUES(12, 'Coffee Shot');
INSERT INTO dbo.Parts(partid, partname) VALUES(13, 'Tea Leaves');
INSERT INTO dbo.Parts(partid, partname) VALUES(14, 'Water');
INSERT INTO dbo.Parts(partid, partname) VALUES(15, 'Sugar Bag');
INSERT INTO dbo.Parts(partid, partname) VALUES(16, 'Ground Coffee');
INSERT INTO dbo.Parts(partid, partname) VALUES(17, 'Coffee Beans');

CREATE TABLE dbo.BOM
(
  partid      INT           NOT NULL REFERENCES dbo.Parts,
  assemblyid  INT           NULL     REFERENCES dbo.Parts,
  unit        VARCHAR(3)    NOT NULL,
  qty         DECIMAL(8, 2) NOT NULL,
  UNIQUE(partid, assemblyid),
  CHECK (partid <> assemblyid)
);

INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 1, NULL, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 2, NULL, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 3, NULL, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 4, NULL, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 5, NULL, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 6,    1, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 7,    1, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(10,    1, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14,    1, 'mL', 230.00);
```

```
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 6,    2, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 7,    2, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(10,    2, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14,    2, 'mL', 205.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(11,    2, 'mL',  25.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 6,    3, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 7,    3, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(11,    3, 'mL', 225.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(12,    3, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 9,    4, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(12,    4, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES( 9,    5, 'EA',   1.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(12,    5, 'EA',   2.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(13,   10, 'g' ,   5.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14,   10, 'mL',  20.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(14,   12, 'mL',  20.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(16,   12, 'g' ,  15.00);
INSERT INTO dbo.BOM(partid, assemblyid, unit, qty)
  VALUES(17,   16, 'g' ,  15.00);
```

טבלה 9–2: תוכן טבלת Parts

partid	*partname*
1	Black Tea
2	White Tea
3	Latte
4	Espresso
5	Double Espresso
6	Cup Cover
7	Regular Cup
8	Stirrer
9	Espresso Cup
10	Tea Shot
11	Milk
12	Coffee Shot
13	Tea Leaves
14	Water
15	Sugar Bag
16	Ground Coffee
17	Coffee Beans

טבלה 9–3: תוכן טבלת BOM

partid	*assemblyid*	*unit*	*qty*
1	NULL	EA	1.00
2	NULL	EA	1.00
3	NULL	EA	1.00
4	NULL	EA	1.00
5	NULL	EA	1.00
6	1	EA	1.00
7	1	EA	1.00

partid	*assemblyid*	*unit*	*qty*
10	1	EA	1.00
14	1	mL	230.00
6	2	EA	1.00
7	2	EA	1.00
10	2	EA	1.00
14	2	mL	205.00
11	2	mL	25.00
6	3	EA	1.00
7	3	EA	1.00
11	3	mL	225.00
12	3	EA	1.00
9	4	EA	1.00
12	4	EA	1.00
9	5	EA	1.00
12	5	EA	2.00
13	10	g	5.00
14	10	mL	20.00
14	12	mL	20.00
16	12	g	15.00
17	16	g	15.00

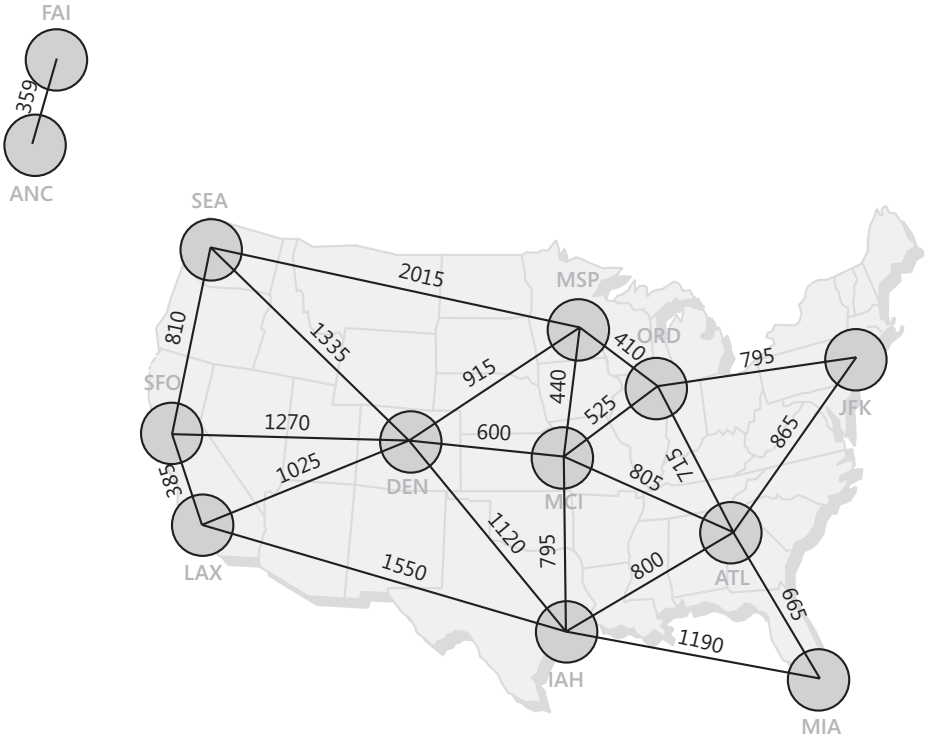
עץ מוצר מייצג גרף לא–מעגלי מכוון (DAG). הוא מחזיק את קודי הצמתים הורה ובן בטורים assemblyid ו–partid, בהתאמה. עץ מוצר מייצג גם גרף **משוקלל**, כאשר משקל/ערך מקושר לכל מקצוע. במקרה שלנו, משקל זה הוא הטור qty המכיל את הכמויות של הפריט בתוך המוצר (מוצר המכיל פריטים). הטור unit מכיל את יחידת המידה של הכמות (EA עבור יחידה, g עבור גרם, mL עבור מיליליטר, וכך הלאה).

מערכת כבישים

מערכת הכבישים בה אשתמש היא זו של מספר ערים מרכזיות בארה"ב, והיא מוצגת גרפית בתרשים 9–4.

כדי ליצור את הטבלאות Cities ו–Roads ולמלא אותן בנתונים לדוגמה, הרץ את הקוד בקטע–קוד 9–3. התוכן של טבלאות Cities ו–Roads מוצג בטבלאות 9–4 ו–9–5.

תרשים 4-9: מערכת כבישים



קטע-קוד 3-9: מבנה ונתונים לדוגמה של הטבלאות Cities ו-Roads

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Roads') IS NOT NULL
  DROP TABLE dbo.Roads;
GO
IF OBJECT_ID('dbo.Cities') IS NOT NULL
  DROP TABLE dbo.Cities;
GO

CREATE TABLE dbo.Cities
(
  cityid   CHAR(3)     NOT NULL PRIMARY KEY,
  city     VARCHAR(30) NOT NULL,
  region   VARCHAR(30) NULL,
  country  VARCHAR(30) NOT NULL
);
```

```
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('ATL', 'Atlanta', 'GA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('ORD', 'Chicago', 'IL', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('DEN', 'Denver', 'CO', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('IAH', 'Houston', 'TX', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('MCI', 'Kansas City', 'KS', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('LAX', 'Los Angeles', 'CA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('MIA', 'Miami', 'FL', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('MSP', 'Minneapolis', 'MN', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('JFK', 'New York', 'NY', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('SEA', 'Seattle', 'WA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('SFO', 'San Francisco', 'CA', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('ANC', 'Anchorage', 'AK', 'USA');
INSERT INTO dbo.Cities(cityid, city, region, country)
  VALUES('FAI', 'Fairbanks', 'AK', 'USA');

CREATE TABLE dbo.Roads
(
  city1     CHAR(3) NOT NULL REFERENCES dbo.Cities,
  city2     CHAR(3) NOT NULL REFERENCES dbo.Cities,
  distance  INT     NOT NULL,
  PRIMARY KEY(city1, city2),
  CHECK(city1 < city2),
  CHECK(distance > 0)
);

INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ANC', 'FAI', 359);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'ORD', 715);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'IAH', 800);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'MCI', 805);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'MIA', 665);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('ATL', 'JFK', 865);
```

```
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'IAH', 1120);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'MCI', 600);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'LAX', 1025);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'MSP', 915);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'SEA', 1335);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('DEN', 'SFO', 1270);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('IAH', 'MCI', 795);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('IAH', 'LAX', 1550);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('IAH', 'MIA', 1190);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('JFK', 'ORD', 795);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('LAX', 'SFO', 385);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MCI', 'ORD', 525);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MCI', 'MSP', 440);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MSP', 'ORD', 410);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('MSP', 'SEA', 2015);
INSERT INTO dbo.Roads(city1, city2, distance) VALUES('SEA', 'SFO', 815);
```

טבלה 4–9: תוכן של טבלת Cities

cityid	*city*	*region*	*country*
ANC	Anchorage	AK	USA
ATL	Atlanta	GA	USA
DEN	Denver	CO	USA
FAI	Fairbanks	AK	USA
IAH	Houston	TX	USA
JFK	New York	NY	USA
LAX	Los Angeles	CA	USA
MCI	Kansas City	KS	USA
MIA	Miami	FL	USA
MSP	Minneapolis	MN	USA
ORD	Chicago	IL	USA
SEA	Seattle	WA	USA
SFO	San Francisco	CA	USA

טבלה 5–9: תוכן של טבלת Roads

city1	*city2*	*distance*
ANC	FAI	359
ATL	IAH	800
ATL	JFK	865
ATL	MCI	805
ATL	MIA	665
ATL	ORD	715
DEN	IAH	1120
DEN	LAX	1025
DEN	MCI	600
DEN	MSP	915
DEN	SEA	1335
DEN	SFO	1270
IAH	LAX	1550
IAH	MCI	795
IAH	MIA	1190
JFK	ORD	795
LAX	SFO	385
MCI	MSP	440
MCI	ORD	525
MSP	ORD	410
MSP	SEA	2015
SEA	SFO	815

טבלת Roads מייצגת גרף בלתי–מכוון מעגלי משוקלל. כל מקצוע (road) מיוצג על ידי שורה בטבלה. הטורים city1 ו–city2 הם שני קודי עיר המייצגים את הצמתים של מקצוע. המשקל במקרה זה הוא הטור distance, המכיל את המרחק בין שתי הערים במיילים. שים לב שבטבלת Roads יש אילוץ CHECK (city1 < city2) כחלק מהגדרת הסכמה שלה, כדי למנוע ניסיונות להוסיף את אותו מקצוע פעמיים (לדוגמה, {SEA, SFO} ו– {SFO, SEA}).

כעת כשבידנו כל התרחישים והנתונים לדוגמה, הבה נעבור על הגישות לטיפול בגרפים, עצים והיררכיות. אדון בשלוש גישות עיקריות: איטרציה/רקורסיה, מסלול ממומש (materialized path) וסטים מקוננים (nested sets).

איטרציה/רקורסיה

גישות איטרטיביות מפעילות צורה כלשהי של לולאות או רקורסיה. קיימים אלגוריתמים איטרטיביים רבים לסריקת גרפים. חלקם סורקים גרפים צומת אחד בכל פעם ולרוב מיושמים עם סמנים, אך אלו לרוב איטיים מאוד. אני אתמקד באלגוריתמים הסורקים גרפים רמה בכל פעם, תוך שימוש בשילוב של לוגיקה איטרטיבית או רקורסיבית ושאילתות מבוססות–סטים. בהינתן סט של צמתים U, **הרמה הבאה של כפופים** היא הסט V, המורכב מהכפופים הישירים (בנים) של הצמתים ב–U. מניסיוני, ליישומים של אלגוריתמים איטרטיביים הסורקים גרף רמה בכל פעם, ביצועים טובים הרבה יותר מאלו הסורקים גרף צומת בכל פעם.

קיימים מספר יתרונות לשימוש בפתרונות איטרטיביים על פני שיטות אחרות. ראשית, אינך צריך להחזיק במסד הנתונים כל מידע נוסף המתאר את הגרף, מלבד קודי הצמתים במקצועות. במילים אחרות, אין צורך לעצב מחדש את הטבלאות שלך. הפתרונות סורקים את הגרף על ידי הסתמכות אך ורק על מידע המקצוע המאוחסן – למשל, (mgrid, empid), (assemblyid, partid), (city1, city2), וכך הלאה.

שנית, מרבית הפתרונות המתאימים לעצים, מתאימים גם לגרפים המכוונים (digraphs) הכלליים יותר. במילים אחרות, מרבית הפתרונות המתאימים לגרפים בהם מסלול אחד בלבד יכול להוביל לצומת נתון, מתאימים גם לגרפים בהם מספר מסלולים עשויים להוביל לצומת נתון.

לבסוף, מרבית הפתרונות שאתאר בסעיף זה, תומכים למעשה במספר בלתי מוגבל של רמות.

בדוגמאות שלי אשתמש בשני כלים עיקריים ליישום פתרונות: פונקציות מוגדרות–משתמש (UDFs) וביטויי טבלה שגורים רקורסיביים (CTEs). UDFs היו זמינים מאז SQL Server 2000, בעוד ש–CTEs הוצגו ב– SQL Server 2005. כאשר אשתמש ב–UDFs, אסתמך רק על מאפיינים תומכי SQL Server 2000 כך שתוכל ליישם את הפתרונות ב– SQL Server 2000. מכיוון ש–CTEs חדשים ב– SQL Server 2005, הרגשתי חופשי להסתמך על מאפיינים אחרים חדשים של T-SQL (למשל, שימוש בפונקציה ROW_NUMBER). האלגוריתמים המרכזיים יהיו דומים בשתי הגרסאות.

בפתרונותי אתמקד ב–UDFs וב–CTEs, אך שים לב שישנם מקרים בהם הביצועים של UDF או CTE אינם משביעי רצון, ייתכן שתקבל ביצועים טובים יותר על ידי יישום פתרון עם פרוצדורה מאוחסנת. פרוצדורות מאוחסנות נותנות לך שליטה רבה יותר – למשל, אתה יכול להחזיק סטי ביניים בטבלאות זמניות ולייצר עליהם אינדקסים, וכדומה.

על כל פנים, השתמשתי ב–UDFs וב–CTEs מכיוון שרציתי להתמקד באלגוריתמים ובבהירות הפתרון.

כפופים

הבא נתחיל בבקשה קלאסית להחזרת כפופים (subordinates); למשל, החזר את כל הכפופים של עובד נתון. טכנית יותר, אתה מעוניין בתת–גרף/תת–עץ של שורש נתון בגרף מכוון. האלגוריתם האיטרטיבי פשוט מאוד:

קלט: @root

אלגוריתם:

- הצב @lvl = 0; הכנס לטבלה @Subs שורה עבור @root
- כל עוד קיימות שורות ברמה הקודמת של העובדים:
  - הצב @lvl = @lvl + 1; הכנס לטבלה @Subs שורות עבור הרמה הבאה (mgrid מופיע ב–(ערכים של empid ברמה הקודמת)).
  - החזר את @Subs.

הרץ את הקוד בקטע–קוד 4–9 כדי ליצור את הפונקציה fn_subordinates1, המיישמת אלגוריתם זה כ–UDF.

קטע–קוד 4–9: קוד יצירה של הפונקציה fn_subordinates1

```
-----------------------------------------------------------------------
-- Function: fn_subordinates1, Descendants
--
-- Input   : @root INT: Manager id
--
-- Output  : @Subs Table: id and level of subordinates of
--                        input manager (empid = @root) in all levels
--
-- Process : * Insert into @Subs row of input manager
--           * In a loop, while previous insert loaded more than 0 rows
--             insert into @Subs next level of subordinates
-----------------------------------------------------------------------
 USE tempdb;
 GO
 IF OBJECT_ID('dbo.fn_subordinates1') IS NOT NULL
   DROP FUNCTION dbo.fn_subordinates1;
 GO
 CREATE FUNCTION dbo.fn_subordinates1(@root AS INT) RETURNS @Subs TABLE
 (
   empid INT NOT NULL PRIMARY KEY NONCLUSTERED,
   lvl   INT NOT NULL,
```

```
  UNIQUE CLUSTERED(lvl, empid)      -- Index will be used to filter
level
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;                     -- Initialize level counter with 0

  -- Insert root node into @Subs
  INSERT INTO @Subs(empid, lvl)
    SELECT empid, @lvl FROM dbo.Employees WHERE empid = @root;

  WHILE @@rowcount > 0              -- while previous level had rows
  BEGIN
    SET @lvl = @lvl + 1;            -- Increment level counter

    -- Insert next level of subordinates to @Subs
    INSERT INTO @Subs(empid, lvl)
      SELECT C.empid, @lvl
      FROM @Subs AS P               -- P = Parent
        JOIN dbo.Employees AS C     -- C = Child
          ON P.lvl = @lvl - 1       -- Filter parents from previous level
          AND C.mgrid = P.empid;
  END

  RETURN;
END
GO
```

הפונקציה מקבלת את פרמטר הקלט @root, שהוא הקוד של עובד השורש של תת–העץ המבוקש. הפונקציה מחזירה את משתנה הטבלה @Subs, עם כל הכפופים של עובד בעל ID = @root בכל הרמות. מלבד כל טורי העובד, קיים ב–@Subs גם טור הנקרא lvl השומר את הרמה בתת–העץ (0 עבור שורש תת–העץ, וגדל משם ב–1 בכל איטרציה).

הקוד של הפונקציה שומר את הרמה הנוכחית שמעובדת במשתנה המקומי @lvl, המאותחל עם אפס.

קוד הפונקציה ראשית מכניס לתוך @Subs את השורה מ–Employees שבה empid = @root.

אז בלולאה, כל עוד ההוספה האחרונה השפיעה על יותר מאפס שורות, הקוד מגדיל את ערך המשתנה @lvl באחד וטוען ל–@Subs את רמת העובדים הבאה – במילים אחרות, כפופים ישירים של המנהלים ברמה הקודמת.

כדי לטעון את רמת העובדים הבאה ל–Subs@, השאילתה בלולאה מבצעת join בין Subs@ (המייצגת מנהלים) לבין Employees (המייצגת כפופים).

הטור lvl חשוב מכיוון שהוא מאפשר לך לבודד את המנהלים שהוכנסו ל–Subs@ באיטרציה האחרונה. כדי להחזיר אך ורק כפופים של המנהלים שהוכנסו קודם, תנאי ה–join מסנן מ–Subs@ רק שורות בהן הטור lvl שווה לרמה הקודמת (lvl - 1@).

כדי לבדוק את הפונקציה, הרץ את הקוד הבא, המחזיר את הכפופים של עובד 3, כפי שמוצג בטבלה 6–9:

```
SELECT empid, lvl FROM dbo.fn_subordinates1(3) AS S;
```

טבלה 6–9: תת–עץ של עובד 3, קודי עובד בלבד

empid	*lvl*
3	0
7	1
9	2
11	2
12	3
13	3
14	3

תוכל לוודא שהפלט נכון על ידי התבוננות בתרשים 2–9 ובחינת תת–העץ של עובד השורש (ID = 3).

כדי לקבל מאפיינים אחרים של העובד מלבד קוד העובד, תוכל לכתוב את הפונקציה שוב ולהוסיף את הטורים הללו לטבלת Subs@, או שתוכל פשוט לבצע join בין הפונקציה לטבלת Employees כלהלן:

```
SELECT E.empid, E.empname, S.lvl
FROM dbo.fn_subordinates1(3) AS S
  JOIN dbo.Employees AS E
    ON E.empid = S.empid;
```

תקבל את הפלט המוצג בטבלה 7–9.

טבלה 7–9: כפופים של עובד 3, כולל שמות עובדים

empid	*empname*	*lvl*
3	Ina	0
7	Aaron	1
9	Rita	2
11	Gabriel	2
12	Emilia	3
13	Michael	3
14	Didi	3

כדי להגביל את סט התוצאה לעובדים ברמת העלה תחת שורש נתון, הוסף פשוט מסנן עם הביטוי NOT EXISTS, כדי לבחור רק עובדים שאינם מנהלים של עובדים אחרים:

```
SELECT empid
FROM dbo.fn_subordinates1(3) AS P
WHERE NOT EXISTS
  (SELECT * FROM dbo.Employees AS C
   WHERE c.mgrid = P.empid);
```

שאילתה זו מחזירה את קודי עובד 11, 12, 13 ו–14.

עד כה, ראית את יישום ה–UDF של תת–עץ תחת שורש נתון. קטע–קוד 5–9 מביא את פתרון ה–CTE (SQL Server 2005 בלבד).

קטע–קוד 5–9: תת–עץ של שורש נתון, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 3;

WITH SubsCTE
AS
(
  -- Anchor member returns root node
  SELECT empid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  -- Recursive member returns next level of children
```

```
  SELECT C.empid, C.empname, P.lvl + 1
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT * FROM SubsCTE;
```

הרצת הקוד בקטע–קוד 5–9 נותנת לך את אותה תוצאה המוצגת בטבלה 7–9.

הפתרון מפעיל לוגיקה דומה לזו שביישום ה–UDF. הוא פשוט יותר במובן שאינך צריך להגדיר מפורשות את הטבלה המוחזרת או לסנן את מנהלי הרמה הקודמת.

השאילתה הראשונה בגוף ה–CTE מחזירה את השורה עבור עובד השורש הנתון מ–Employees. היא גם מחזירה אפס בטור lvl עבור עובד השורש. ב–CTE רקורסיבי, שאילתה שאין לה כל פניות רקורסיביות ידועה גם **כאיבר עוגן** (anchor member).

לשאילתה השנייה בגוף ה–CTE (לאחר פעולת הסט UNION ALL) יש פנייה רקורסיבית לשם ה–CTE. דבר זה הופך אותה **לאיבר רקורסיבי** (recursive member), והיא מטופלת באופן מיוחד. הפנייה הרקורסיבית לשם ה–CTE (SubsCTE) מייצגת את סט התוצאה שהוחזר קודם. שאילתת האיבר הרקורסיבי מבצעת join בין סט התוצאה הקודם, המייצג את המנהלים ברמה הקודמת, לבין טבלת Employees כדי להחזיר את רמת העובדים הבאה. השאילתה הרקורסיבית מחשבת גם את ערך lvl כרמת המנהל של העובד ועוד אחד. בפעם הראשונה שהאיבר הרקורסיבי מופעל, SubsCTE מייצג את סט התוצאה המוחזר מאיבר העוגן (עובד שורש). לא קיימת בדיקת סיום מפורשת עבור האיבר הרקורסיבי; אלא, הוא מופעל שוב ושוב עד שהוא מחזיר סט ריק. כך, בפעם הראשונה שהוא מופעל, הוא מחזיר כפופים ישירים של עובד שורש תת–העץ. בפעם השנייה שהוא מופעל, SubsCTE מייצג את סט התוצאה של ההפעלה הראשונה של האיבר הרקורסיבי (רמה ראשונה של כפופים), כך שהוא מחזיר את הרמה השנייה של כפופים. האיבר הרקורסיבי מופעל שוב ושוב עד שאין יותר כפופים, במקרה כזה הוא יחזיר סט ריק והרקורסיה תיפסק.

הפנייה לשם ה–CTE בשאילתה החיצונית מייצגת את ה– UNION ALL של כל סטי התוצאה המוחזרים מההפעלה של איבר העוגן ומכל ההפעלות של האיבר הרקורסיבי.

כפי שהזכרתי קודם, שימוש בלוגיקה איטרטיבית כדי להחזיר תת–גרף של גרף–מכוון, כאשר עשויים להתקיים מספר מסלולים לצומת, דומה להחזרת תת–עץ. הרץ את הקוד בקטע–קוד 6–9, כדי ליצור את הפונקציה fn_partsexplosion. הפונקציה מקבלת קוד פריט המייצג מוצר בעץ מוצר, ומחזירה את פיצוץ העץ (פריטים ישירים ובלתי–ישירים) של המוצר.

קטע-קוד 6-9: קוד יצירה של הפונקציה fn_partsexplosion

```
-----------------------------------------------------------------------
-- Function: fn_partsexplosion, Parts Explosion
--
-- Input   : @root INT: assembly id
--
-- Output  : @PartsExplosion Table:
--              id and level of contained parts of input part
--              in all levels
--
-- Process : * Insert into @PartsExplosion row of input root part
--           * In a loop, while previous insert loaded more than 0 rows
--             insert into @PartsExplosion next level of parts
-----------------------------------------------------------------------
 USE tempdb;
 GO
 IF OBJECT_ID('dbo.fn_partsexplosion') IS NOT NULL
   DROP FUNCTION dbo.fn_partsexplosion;
 GO
 CREATE FUNCTION dbo.fn_partsexplosion(@root AS INT)
   RETURNS @PartsExplosion Table
 (
   partid INT           NOT NULL,
   qty    DECIMAL(8, 2) NOT NULL,
   unit   VARCHAR(3)    NOT NULL,
   lvl    INT           NOT NULL,
   n      INT           NOT NULL IDENTITY, -- surrogate key
   UNIQUE CLUSTERED(lvl, n)  -- Index will be used to filter lvl
 )
 AS
 BEGIN
   DECLARE @lvl AS INT;
   SET @lvl = 0;                     -- Initialize level counter with 0

   -- Insert root node to @PartsExplosion
   INSERT INTO @PartsExplosion(partid, qty, unit, lvl)
     SELECT partid, qty, unit, @lvl
     FROM dbo.BOM
     WHERE partid = @root;

   WHILE @@rowcount > 0      -- while previous level had rows
   BEGIN
```

```
    SET @lvl = @lvl + 1;            -- Increment level counter

    -- Insert next level of subordinates to @PartsExplosion
    INSERT INTO @PartsExplosion(partid, qty, unit, lvl)
      SELECT C.partid, P.qty * C.qty, C.unit, @lvl
      FROM @PartsExplosion AS P     -- P = Parent
        JOIN dbo.BOM AS C           -- C = Child
          ON P.lvl = @lvl - 1       -- Filter parents from previous level
          AND C.assemblyid = P.partid;
  END

  RETURN;
END
GO
```

היישום של הפונקציה fn_partexplosion דומה ליישום של הפונקציה fn_subordinates1. השורה עבור פריט השורש נטענת למשתנה הטבלה @PartsExplosion (פרמטר הפלט של הפונקציה). ואז בלולאה, כל עוד ההוספה הקודמת טענה יותר מאפס שורות, הפריטים של הרמה הבאה נטענים ל–@PartsExplosion. קיימת כאן תוספת קטנה הספציפית לעץ מוצר – חישוב הכמויות. כמות פריט השורש היא זו המאוחסנת בשורת הפריט. כמות הפריט המוכל (בן) היא כמות הפריט המכיל (הורה) מוכפלת בכמות הפריט המוכל.

כדי לבדוק את הפונקציה הרץ את הקוד הבא, המחזיר את פיצוץ העץ של קוד פריט 2 (White Tea):

```
SELECT P.partid, P.partname, PE.qty, PE.unit, PE.lvl
FROM dbo.fn_partsexplosion(2) AS PE
  JOIN dbo.Parts AS P
    ON P.partid = PE.partid;
```

תוכל לבדוק את נכונות הפלט המוצג בטבלה 8–9 על ידי בחינת תרשים 3–9.

טבלה 8–9: פיצוץ פריט 2

partid	*Partname*	*qty*	*unit*	*lvl*
2	White Tea	1.00	EA	0
6	Cup Cover	1.00	EA	1
7	Regular Cup	1.00	EA	1
10	Tea Shot	1.00	EA	1

partid	Partname	qty	unit	lvl
11	Milk	25.00	mL	1
14	Water	205.00	mL	1
13	Tea Leaves	5.00	g	2
14	Water	20.00	mL	2

בקטע–קוד 7–9, יש את פתרון ה–CTE עבור פיצוץ עץ, אשר שוב, דומה לפתרון תת–העץ עם תוספת חישובי הכמויות.

קטע–קוד 7–9: פתרון CTE עבור פיצוץ עץ

```
DECLARE @root AS INT;
SET @root = 2;

WITH PartsExplosionCTE
AS
(
  -- Anchor member returns root part
  SELECT partid, qty, unit, 0 AS lvl
  FROM dbo.BOM
  WHERE partid = @root

  UNION ALL

  -- Recursive member returns next level of parts
  SELECT C.partid, CAST(P.qty * C.qty AS DECIMAL(8, 2)),
    C.unit, P.lvl + 1
  FROM PartsExplosionCTE AS P
    JOIN dbo.BOM AS C
      ON C.assemblyid = P.partid
)
SELECT P.partid, P.partname, PE.qty, PE.unit, PE.lvl
FROM PartsExplosionCTE AS PE
  JOIN dbo.Parts AS P
    ON P.partid = PE.partid;
```

פיצוץ עץ עשוי להכיל יותר ממופע אחד של אותו פריט מכיוון שפריטים שונים במוצר עשויים לכלול את אותו תת–פריט. לדוגמה, תוכל לראות בטבלה 8–9 שמים מופיעים פעמיים מכיוון שתה לבן מכיל 205 מיליליטר מים ישירות, והוא מכיל גם Tea Shot,

שבתורו מכיל 20 מיליליטר מים. ייתכן שתרצה לקבץ את סט התוצאה לפי פריט ויחידה כלהלן, ולייצר את הפלט המופיע בטבלה 9–9:

```
SELECT P.partid, P.partname, PES.qty, PES.unit
FROM (SELECT partid, unit, SUM(qty) AS qty
      FROM dbo.fn_partsexplosion(2) AS PE
      GROUP BY partid, unit) AS PES
  JOIN dbo.Parts AS P
    ON P.partid = PES.partid;
```

טבלה 9–9: פיצוץ פריט 2, עם פריטים מקובצים

partid	*partname*	*qty*	*unit*
2	White Tea	1.00	EA
6	Cup Cover	1.00	EA
7	Regular Cup	1.00	EA
10	Tea Shot	1.00	EA
13	Tea Leaves	5.00	g
11	Milk	25.00	mL
14	Water	225.00	mL

לא אכנס כאן לסוגיות קיבוץ פריטים אשר עשויים לכלול יחידות מידה שונות. כמובן, תצטרך להתמודד עם אלה על ידי הפעלת גורמי המרה.

כדוגמה נוספת, הקוד הבא מפוצץ את פריט 5 (Double Espresso), ומחזיר את הפלט המוצג בטבלה 10–9:

```
SELECT P.partid, P.partname, PES.qty, PES.unit
FROM (SELECT partid, unit, SUM(qty) AS qty
      FROM dbo.fn_partsexplosion(5) AS PE
      GROUP BY partid, unit) AS PES
  JOIN dbo.Parts AS P
    ON P.partid = PES.partid;
```

טבלה 10–9: פיצוץ פריט 5, עם פריטים מקובצים

partid	*partname*	*qty*	*unit*
5	Double Espresso	1.00	EA
9	Espresso Cup	1.00	EA

partid	*partname*	*qty*	*unit*
12	Coffee Shot	2.00	EA
16	Ground Coffee	30.00	g
17	Coffee Beans	450.00	g
14	Water	40.00	mL

אם ניזכר בהחזרת תת–עץ של עובד נתון, ייתכן שתצטרך במקרים מסוימים להגביל את מספר הרמות המוחזרות. כדי להשיג זאת, קיימת תוספת קטנה שעליך להוסיף לאלגוריתם המקורי:

קלט: @root, @maxlevels (כמות רמות מקסימלית מלבד השורש)

אלגוריתם:

- הצב @lvl = 0; טען לטבלה @Subs שורה עבור @root.
- כל עוד קיימות שורות ברמה הקודמת, וגם @lvl < @maxlevels:
  - הצב @lvl = @lvl + 1; טען לטבלה @Subs שורות עבור הרמה הבאה (mgrid מופיע ב–(ערכי empid ברמה הקודמת)).
  - החזר את @Subs.

הרץ את הקוד בקטע–קוד 8–9 כדי ליצור את הפונקציה fn_subordinates2, שהיא עדכון של fn_subordinates1 שתומכת גם היא בהגבלת רמות.

קטע–קוד 8–9: קוד יצירה של הפונקציה fn_subordinates2

```
-----------------------------------------------------------------------
-- Function: fn_subordinates2,
--           Descendants with optional level limit
--
-- Input  :  @root      INT: Manager id
--           @maxlevels INT: Max number of levels to return
--
-- Output  : @Subs TABLE: id and level of subordinates of
--                        input manager in all levels <= @maxlevels
--
-- Process : * Insert into @Subs row of input manager
--           * In a loop, while previous insert loaded more than 0 rows
--             and previous level is smaller than @maxlevels
--             insert into @Subs next level of subordinates
-----------------------------------------------------------------------
 USE tempdb;
```

```
GO
IF OBJECT_ID('dbo.fn_subordinates2') IS NOT NULL
  DROP FUNCTION dbo.fn_subordinates2;
GO
CREATE FUNCTION dbo.fn_subordinates2
  (@root AS INT, @maxlevels AS INT = NULL) RETURNS @Subs TABLE
(
  empidINT NOT NULL PRIMARY KEY NONCLUSTERED,
  lvl  INT NOT NULL,
  UNIQUE CLUSTERED(lvl, empid)  -- Index will be used to filter level
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;                 -- Initialize level counter with 0
  -- If input @maxlevels is NULL, set it to maximum integer
  -- to virtually have no limit on levels
  SET @maxlevels = COALESCE(@maxlevels, 2147483647);

  -- Insert root node to @Subs
  INSERT INTO @Subs(empid, lvl)
    SELECT empid, @lvl FROM dbo.Employees WHERE empid = @root;

  WHILE @@rowcount > 0          -- while previous level had rows
    AND @lvl < @maxlevels       -- and previous level < @maxlevels
  BEGIN
    SET @lvl = @lvl + 1;        -- Increment level counter

    -- Insert next level of subordinates to @Subs
    INSERT INTO @Subs(empid, lvl)
      SELECT C.empid, @lvl
      FROM @Subs AS P           -- P = Parent
        JOIN dbo.Employees AS C -- C = Child
          ON P.lvl = @lvl - 1   -- Filter parents from previous level
          AND C.mgrid = P.empid;
  END

  RETURN;
END
GO
```

בנוסף לקלט המקורי, fn_subordinates2 מקבלת גם את הקלט @maxlevels המציין את מספר הרמות המקסימלי שיש להחזיר תחת @root. כדי לא להציב הגבלה על מספר הרמות, הפונקציה מחליפה את ה–NULL עם ערך השלם המקסימלי האפשרי, כך שלמעשה לא תהיה הגבלה.

תנאי הלולאה, מלבד ביצוע בדיקה שההוספה הקודמת השפיעה על יותר מאפס שורות, בודק גם שהמשתנה @lvl קטן מ–@maxlevels. מלבד שינויים קלים אלו, יישום הפונקציה זהה לזה של fn_subordinates1.

כדי לבדוק את הפונקציה, הרץ את הקוד הבא המבקש כפופים של עובד 3 בכל הרמות (@maxlevels הוא NULL) ומפיק את הפלט המוצג בטבלה 9–11:

```
SELECT empid, lvl
FROM dbo.fn_subordinates2(3, NULL) AS S;
```

טבלה 9–11: תת–עץ של עובד 3, ללא הגבלת רמות

empid	*lvl*
3	0
7	1
9	2
11	2
12	3
13	3
14	3

כדי לקבל שתי רמות כפופים בלבד תחת עובד 3, הרץ את הקוד הבא, המפיק את הפלט המוצג בטבלה 9-12:

```
SELECT empid, lvl
FROM dbo.fn_subordinates2(3, 2) AS S;
```

טבלה 9–12: תת–עץ של עובד 3, עד 2 רמות

empid	*lvl*
3	0
7	1
9	2
11	2

כדי לקבל רק את רמת העובדים השנייה תחת עובד 3, הוסף מסנן על הרמה, וזה יפיק את הפלט המוצג בטבלה 13–9:

```
SELECT empid
FROM dbo.fn_subordinates2(3, 2) AS S
WHERE lvl = 2;
```

טבלה 13–9: תת–עץ של עובד 3, רמה 2 בלבד

empid
9
11

אזהרה: כדי להגביל רמות בעת שימוש ב–CTE, ייתכן שתתפתה להשתמש ב–hint שנקרא MAXRECURSION, המחזיר הודעת שגיאה ומפסיק את פעילות השאילתה, כאשר מספר הקריאות לאיבר הרקורסיבי עוברות את הקלט. עם זאת, MAXRECURSION עוצב כאמצעי ביטחון כדי להימנע מרקורסיה אינסופית במקרים של בעיה בנתונים או באגים בקוד. כאשר אינו מוגדר אחרת, ברירת המחדל של MAXRECURSION היא 100. תוכל להגדיר את ה–MAXRECURSION כ–0 כך שלא תהיה מגבלה, אך עליך להיות מודע להשלכות.

כדי לבדוק גישה זו, הרץ את הקוד בקטע–קוד 9–9, המפיק את הפלט המוצג בטבלה 14–9. זהו אותו CTE של תת–העץ שהוצג קודם, עם התוספת של ה– MAXRECURSION hint, המגביל את הקריאה הרקורסיבית ל–2.

קטע–קוד 9–9: תת–עץ עם הגבלת רמות, פתרון CTE עם MAXRECURSION

```
DECLARE @root AS INT;
SET @root = 3;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL
```

```
  SELECT C.empid, C.empname, P.lvl + 1
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT * FROM SubsCTE
OPTION (MAXRECURSION 2);
```

טבלה 14–9: תת–עץ של עובד 3, רמות מוגבלות על ידי MAXRECURSION

empid	*empname*	*lvl*
3	Ina	0
7	Aaron	1
11	Gabriel	2
9	Rita	2

```
Server: Msg 530, Level 16, State 1, Line 4
```

אזהרה: המשפט הופסק. הרקורסיה הופעלה יותר מפעמיים בטרם הסתיימה השאילתה.

הקוד נשבר ברגע שהאיבר הרקורסיבי מופעל בפעם השלישית. אין זה מומלץ להשתמש ב– MAXRECURSION hint כדי להגביל לוגית את מספר הרמות משתי סיבות: ראשית, הודעת שגיאה מופקת למרות שאין כל שגיאה לוגית כאן. שנית, SQL Server אינו מבטיח להחזיר כל סט תוצאה אם מוחזרת שגיאה. במקרה המסוים הזה, סט התוצאה הוחזר, אך אין ערובה לכך שזה יקרה במקרים אחרים.

כדי להגביל לוגית את מספר הרמות, פשוט הוסף מסנן על טור רמת המנהל בתנאי ה–join של האיבר הרקורסיבי, כפי שמוצג בקטע–קוד 10–9.

קטע–קוד 10–9: תת–עץ עם הגבלת רמות, פתרון CTE, עם טור רמה

```
DECLARE @root AS INT, @maxlevels AS INT;
SET @root = 3;
SET @maxlevels = 2;

WITH SubsCTE
```

```
AS
(
  SELECT empid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
      AND P.lvl < @maxlevels -- limit parent's level
)
SELECT * FROM SubsCTE;
```

אבות (Ancestors)

בקשות לאבות של צומת נתון גם הן שכיחות – למשל, החזרת שרשרת הניהול עבור עובד נתון. שלא במפתיע, האלגוריתמים להחזרת אבות על ידי שימוש בלוגיקה איטרטיבית דומים לאלו המשמשים להחזרת כפופים. פשוט, במקום להתחיל לסרוק את הגרף מצומת מסוים ולהמשיך "כלפי מטה" לצמתים בנים, אתה מתחיל בצומת נתון וממשיך "כלפי מעלה" לצמתי הורה.

הרץ את הקוד בקטע–קוד 11–9 כדי ליצור את הפונקציה fn_managers. הפונקציה מקבלת כקלט קוד עובד (@empid), ואופציונלית, הגבלת רמה (@maxlevels), ומחזירה מנהלים עד למספר הרמות המבוקש מעל עובד הקלט (אם צוינה הגבלה).

קטע–קוד 11–9: קוד יצירה של הפונקציה fn_managers

```
-----------------------------------------------------------------------
-- Function: fn_managers, Ancestors with optional level limit
--
-- Input   : @empid INT : Employee id
--           @maxlevels : Max number of levels to return
--
-- Output  : @Mgrs Table: id and level of managers of
--                        input employee in all levels <= @maxlevels
--
-- Process : * In a loop, while current manager is not null
--             and previous level is smaller than @maxlevels
```

```
--              insert into @Mgrs current manager,
--              and get next level manager
-----------------------------------------------------------------------
 USE tempdb;
 GO
 IF OBJECT_ID('dbo.fn_managers') IS NOT NULL
   DROP FUNCTION dbo.fn_managers;
 GO
 CREATE FUNCTION dbo.fn_managers
   (@empid AS INT, @maxlevels AS INT = NULL) RETURNS @Mgrs TABLE
 (
   empid INT NOT NULL PRIMARY KEY,
   lvl   INT NOT NULL
 )
 AS
 BEGIN
   IF NOT EXISTS(SELECT * FROM dbo.Employees WHERE empid = @empid)
     RETURN;

   DECLARE @lvl AS INT;
   SET @lvl = 0;              -- Initialize level counter with 0
   -- If input @maxlevels is NULL, set it to maximum integer
   -- to virtually have no limit on levels
   SET @maxlevels = COALESCE(@maxlevels, 2147483647);

   WHILE @empid IS NOT NULL  -- while current employee has a manager
     AND @lvl <= @maxlevels  -- and previous level < @maxlevels
   BEGIN
     -- Insert current manager to @Mgrs
     INSERT INTO @Mgrs(empid, lvl) VALUES(@empid, @lvl);
     SET @lvl = @lvl + 1;    -- Increment level counter
     -- Get next level manager
     SET @empid = (SELECT mgrid FROM dbo.Employees
                   WHERE empid = @empid);
   END

   RETURN;
 END
 GO
```

הפונקציה בודקת תחילה האם קוד הצומת של הקלט קיים, ואז יוצאת אם הוא אינו קיים. אז היא מאתחלת את המונה @lvl עם אפס, ומציבה את השלם הגבוה ביותר האפשרי במשתנה @maxlevels אם צוין בו NULL, כך שלמעשה לא תהיה כל הגבלת רמות.

הפונקציה נכנסת אז ללולאה החוזרת על עצמה כל עוד @empid אינו NULL (מכיוון ש–NULL מייצג את קוד המנהל של עובד השורש), והרמה הנוכחית קטנה או שווה למספר הרמות המבוקש. גוף הלולאה טוען את העובד הנוכחי יחד עם מונה הרמות לתוך משתנה הטבלה @Mgrs של הפלט, מגדיל את מונה הרמות, ומציב את קוד המנהל של העובד הנוכחי במשתנה @empid.

עלי לציין מספר הבדלים בין פונקציה זו לפונקציית הכפופים. פונקציה זו משתמשת בתת–שאילתה סקלארית כדי לקבל את קוד המנהל ברמה הבאה, שלא כמו פונקציית הכפופים אשר השתמשה ב–join כדי לקבל את הרמה הבאה של כפופים. הסיבה להבדל היא שיכול להיות רק מנהל אחד לעובד נתון, בעוד שיכולים להיות מספר כפופים למנהל נתון. כמו כן, פונקציה זו משתמשת בביטוי @lvl <= @maxlevels כדי להגביל את מספר הרמות, בעוד שפונקציית הכפופים השתמשה בביטוי @lvl < @maxlevels. הסיבה לשונות היא שלפונקציה זו אין משפט INSERT אחד כדי לקבל את עובד השורש ומשפט אחר כדי לקבל את רמת העובדים הבאה; אלא, יש לה משפט INSERT אחד בלבד בלולאה. כתוצאה מכך, המונה @lvl כאן גדל אחרי ה–INSERT, בעוד שבפונקציית הכפופים הוא גדל לפני ה–INSERT.

כדי לבדוק את הפונקציה, הרץ את הקוד הבא, המחזיר את המנהלים של עובד 8 בכל הרמות ומפיק את הפלט המוצג בטבלה 15–9:

```
SELECT empid, lvl
FROM dbo.fn_managers(8, NULL) AS M;
```

טבלה 15–9: שרשרת ניהול של עובד 8, ללא הגבלת רמות

empid	*lvl*
1	3
2	2
5	1
8	0

פתרון ה–CTE להחזרת אבות זהה כמעט לחלוטין לפתרון ה–CTE המחזיר תת–עץ. ההבדל הקטן הוא שכאן האיבר הרקורסיבי מתייחס לשם ה–CTE כחלק הבן של ה–join ולטבלת Employees כחלק ההורה, בעוד שבפתרון תת–העץ התפקידים היו הפוכים. הרץ את הקוד בקטע–קוד 12–9 כדי לקבל את שרשרת הניהול של עובד 8, תוך שימוש ב–CTE וכדי להפיק את הפלט המוצג בטבלה 16–9.

קטע-קוד 9-12: שרשרת ניהול של עובד 8, פתרון CTE

```
DECLARE @empid AS INT;
SET @empid = 8;

WITH MgrsCTE
AS
(
  SELECT empid, mgrid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @empid

  UNION ALL

  SELECT P.empid, P.mgrid, P.empname, C.lvl + 1
  FROM MgrsCTE AS C
    JOIN dbo.Employees AS P
      ON C.mgrid = P.empid
)
SELECT * FROM MgrsCTE;
```

טבלה 9-16: פלט CTE עבור שרשרת ניהול של עובד 8

empid	*mgrid*	*empname*	*lvl*
8	5	Lilach	0
5	2	Jiru	1
2	1	Eitan	2
1	NULL	David	3

כדי לקבל רק שתי רמות של מנהלים של עובד 8 על ידי הפונקציה fn_managers, הרץ את הקוד הבא, המפיק את הפלט המוצג בטבלה 9-17:

```
SELECT empid, lvl
FROM dbo.fn_managers(8, 2) AS M;
```

טבלה 17–9: שרשרת ניהול של עובד 8, הגבלת 2 רמות, פתרון CTE

empid	*lvl*
2	2
5	1
8	0

וכדי להחזיר רק את המנהל של הרמה השנייה, פשוט הוסף מסנן בשאילתה החיצונית, המחזירה קוד עובד 2:

```
SELECT empid
FROM dbo.fn_managers(8, 2) AS M
WHERE lvl = 2;
```

כדי להחזיר שתי רמות של מנהלים של עובד 8 עם CTE, הוסף מסנן על רמת הבן בתנאי ה–join של האיבר הרקורסיבי, כפי שמוצג בקטע–קוד 13–9.

קטע–קוד 13–9: אבות עם הגבלת רמה, פתרון CTE

```
DECLARE @empid AS INT, @maxlevels AS INT;
SET @empid = 8;
SET @maxlevels = 2;

WITH MgrsCTE
AS
(
  SELECT empid, mgrid, empname, 0 AS lvl
  FROM dbo.Employees
  WHERE empid = @empid

  UNION ALL

  SELECT P.empid, P.mgrid, P.empname, C.lvl + 1
  FROM MgrsCTE AS C
    JOIN dbo.Employees AS P
      ON C.mgrid = P.empid
      AND C.lvl < @maxlevels -- limit child's level
)
SELECT * FROM MgrsCTE;
```

פתרונות תת–גרף/תת–עץ עם מסלול אבות (Path Enumeration)

בפתרונות תת–הגרף/תת–העץ (subgraph/subtree), ייתכן שתרצה גם לייצר לכל צומת מסלול אבות (enumerated path), המורכב מכל קודי הצמתים במסלול המובילים לצומת היעד, מופרדים בצורה כלשהי (למשל, '.'). לדוגמה, מסלול האבות עבור עובד 8 בתרחיש המבנה הארגוני הוא (משמאל לימין) '1.2.5.8.'. מכיוון שעובד 5 הוא המנהל של עובד 8, עובד 2 הוא המנהל של 5, עובד 1 הוא המנהל של 2, והוא גם עובד השורש.

למסלול אבות קיימים שימושים רבים – למשל, כדי למיין בפלט את הצמתים מההיררכיה, כדי לזהות מחזוריות ושימושים נוספים אותם אתאר בהמשך בסעיף "מסלול אבות ממומש". תוכל לערוך שינויים קלים בפתרונות שסיפקתי להחזרת תת–גרף/תת–עץ לחישוב מסלול האבות ללא כל I/O נוסף.

האלגוריתם מתחיל בצומת השורש של תת–העץ, ובלולאה או ברקורסיה מחזיר את הרמה הבאה. עבור צומת השורש, המסלול הוא פשוט: '.' + node id + '.'. עבור צמתים ברמות עוקבות, המסלול הוא: parent's path + node id + '.'.

הרץ את הקוד בקטע–קוד 14–9 כדי ליצור את הפונקציה fn_subordinates3, הזהה לפונקציה fn_subordinates2 מלבד התוספת של חישוב מסלול האבות.

קטע–קוד 14–9: קוד יצירה של הפונקציה fn_subordinates3

```
----------------------------------------------------------------------
-- Function: fn_subordinates3,
--           Descendants with optional level limit,
--           and path enumeration
--
-- Input   :  @root      INT: Manager id
--            @maxlevels INT: Max number of levels to return
--
-- Output  : @Subs TABLE: id, level and materialized ancestors path
--                        of subordinates of input manager
--                        in all levels <= @maxlevels
--
-- Process : * Insert into @Subs row of input manager
--           * In a loop, while previous insert loaded more than 0 rows
--             and previous level is smaller than @maxlevels:
--             - insert into @Subs next level of subordinates
--             - calculate a materialized ancestors path for each
--             by concatenating current node id to parent's path
----------------------------------------------------------------------
```

```
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_subordinates3') IS NOT NULL
  DROP FUNCTION dbo.fn_subordinates3;
GO
CREATE FUNCTION dbo.fn_subordinates3
  (@root AS INT, @maxlevels AS INT = NULL) RETURNS @Subs TABLE
(
  empidINT           NOT NULL PRIMARY KEY NONCLUSTERED,
  lvl  INT           NOT NULL,
  path VARCHAR(900)  NOT NULL
  UNIQUE CLUSTERED(lvl, empid)  -- Index will be used to filter level
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;                 -- Initialize level counter with 0
  -- If input @maxlevels is NULL, set it to maximum integer
  -- to virtually have no limit on levels
  SET @maxlevels = COALESCE(@maxlevels, 2147483647);

  -- Insert root node to @Subs
  INSERT INTO @Subs(empid, lvl, path)
    SELECT empid, @lvl, '.' + CAST(empid AS VARCHAR(10)) + '.'
    FROM dbo.Employees WHERE empid = @root;

  WHILE @@rowcount > 0          -- while previous level had rows
    AND @lvl < @maxlevels       -- and previous level < @maxlevels
  BEGIN
    SET @lvl = @lvl + 1;        -- Increment level counter

    -- Insert next level of subordinates to @Subs
    INSERT INTO @Subs(empid, lvl, path)
      SELECT C.empid, @lvl,
        P.path + CAST(C.empid AS VARCHAR(10)) + '.'
      FROM @Subs AS P           -- P = Parent
        JOIN dbo.Employees AS C -- C = Child
          ON P.lvl = @lvl - 1   -- Filter parents from previous level
          AND C.mgrid = P.empid;
  END
 RETURN;
END
GO
```

כדי לבדוק את הפונקציה, הרץ את הקוד הבא, המחזיר את כל הכפופים של עובד 1 והמסלולים שלהם, כפי שמוצג בטבלה 18–9:

```
SELECT empid, lvl, path
FROM dbo.fn_subordinates3(1, NULL) AS S;
```

טבלה 18–9: תת–עץ עם מסלול אבות

empid	*lvl*	*path*
1	0	.1.
2	1	.1.2.
3	1	.1.3.
4	2	.1.2.4.
5	2	.1.2.5.
6	2	.1.2.6.
7	2	.1.3.7.
8	3	.1.2.5.8.
9	3	.1.3.7.9.
10	3	.1.2.5.10.
11	3	.1.3.7.11.
12	4	.1.3.7.9.12.
13	4	.1.3.7.9.13.
14	4	.1.3.7.9.14.

כשיש בידך את הערכים lvl ו–path, אתה יכול להחזיר בקלות פלט המציג גרפית את היחסים ההיררכיים של העובדים בתת–העץ:

```
SELECT E.empid, REPLICATE(' | ', lvl) + empname AS empname
FROM dbo.fn_subordinates3(1, NULL) AS S
  JOIN dbo.Employees AS E
    ON E.empid = S.empid
ORDER BY path;
```

השאילתה מבצעת join בין תת–העץ המוחזר מהפונקציה fn_subordinates3 לבין טבלת Employees, בהתבסס על התאמת קוד עובד. מהפונקציה, אתה מקבל את הערכים lvl ו–path, ומהטבלה אתה מקבל מאפיינים אחרים של העובד המעניינים אותך, כמו שם העובד. אתה מייצר אינדנטציה (indentation) לפני שם העובד על ידי שכפול מחרוזת

(במקרה זה, '|') lvl פעמים ושרשור שם העובד אליה. מיון העובדים לפי הטור path מייצר מיון היררכי נכון, הדורש שצומת בן יופיע אחרי צומת ההורה שלו – או במילים אחרות, שלצומת בן יהיה ערך מיון גבוה יותר מאשר לצומת ההורה שלו. בהגדרה, מסלול בן גדול ממסלול הורה מכיוון שהתחילית שלו הוא מסלול ההורה. הפלט של שאילתה זו מוצג בטבלה 19–9.

טבלה 19–9: תת–עץ, ממוין לפי מסלול ובאינדנטציה לפי רמה

empid	*empname*
1	David
2	| Eitan
4	| | Seraph
5	| | Jiru
10	| | | Sean
8	| | | Lilach
6	| | Steve
3	| Ina
7	| | Aaron
11	| | | Gabriel
9	| | | Rita
12	| | | | Emilia
13	| | | | Michael
14	| | | | Didi

בדומה, תוכל להוסיף ל–CTE חישוב מסלול בתת–העץ כפי שמוצג בקטע–קוד 15–9.

קטע–קוד 15–9: תת–עץ עם מסלול אבות, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    -- Path of root = '.' + empid + '.'
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
                        AS VARCHAR(MAX)) AS path
```

```
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    -- Path of child = parent's path + child empid + '.'
    CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
          AS VARCHAR(MAX)) AS path
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, REPLICATE(' | ', lvl) + empname AS empname
FROM SubsCTE
ORDER BY path;
```

שים לב: טורים תואמים בין איבר עוגן לאיבר רקורסיבי של CTE חייבים להיות זהים הן בטיפוס הנתונים והן בגודל. זו הסיבה שהמרתי את מחרוזות המסלול בשניהם לאותו טיפוס נתונים וגודל – VARCHAR(MAX).

מיון

מיון (sort) היא בקשת הצגה ומשמשת לרוב לקוחות פחות מאשר את השרת. המשמעות היא שייתכן שתרצה שמיון היררכיות יקרה בלקוח. בסעיף זה, עם זאת, אציג שיטות מיון בצד–השרת עם T-SQL בהן באפשרותך להשתמש כאשר אתה מעדיף לטפל במיון בשרת.

מיון טופולוגי של DAG מוגדר ככזה המספק לבן ערך מיון גבוה יותר מאשר להורה שלו. לעיתים, אתייחס למיון טופולוגי באופן בלתי רשמי כ"מיון היררכי נכון". יותר מדרך מיון אחת של פריטים ב–DAG עשויה להיחשב נכונה. הסדר בין אחים עשוי להיות חשוב לך או שלא. אם הסדר בין אחים אינו חשוב לך, תוכל להשיג מיון על ידי בניית מסלול אבות לכל צומת, כפי שתואר בסעיף הקודם, ומיון הצמתים לפי מסלול זה.

זכור שמסלול האבות הוא מחרוזת תווים הבנויה מקודי האבות המובילים לצומת, על ידי שימוש במפריד כלשהו. המשמעות היא שאחים מסודרים לפי קוד הצומת שלהם. מכיוון שהמסלול הוא מבוסס–תווים, אתה מקבל מיון מבוסס–תווים של קודים, אשר עשוי להיות שונה מאשר ערכם המספרי של הקודים. לדוגמה, קוד עובד 11 ימוין לפני האח שלו בעל קוד עובד 9 ('.1.3.7.9.' < '.1.3.7.11.') למרות ש–9 < 11. אתה יכול להבטיח שמיון לפי

מסלול האבות יפיק מיון היררכי נכון, אך הוא לא יבטיח את הסדר בין אחים. אם דרוש לך ביטחון כזה, אתה זקוק לפתרון אחר.

כדי לקבל גמישות מיון מקסימלית, ייתכן שתרצה להבטיח את הדברים הבאים:

1. מיון טופולוגי נכון – כלומר, מיון שבו לבן יהיה ערך מיון גבוה יותר מאשר זה של ההורה שלו.
2. אחים ממוינים לפי סדר מבוקש (לדוגמה, לפי empname או לפי salary).
3. יצירת ערכי מיון במספרים שלמים, בניגוד למחרוזות ארוכות.

בפתרון מסלול האבות, דרישה 1 מתמלאת. דרישה 2 אינה מתמלאת מכיוון שהמסלול עשוי מקודי צומת והוא מבוסס–תווים; השוואה ומיון בין תווים מבוססים על מאפייני collation, ומפיקים התנהגות השוואה ומיון שונה מאשר עם מספרים שלמים. דרישה 3 אינה מתמלאת מכיוון שהפתרון מסדר את התוצאות לפי המסלול, שהוא ארוך יותר בהשוואה לערך מספר שלם. כדי למלא את כל שלוש הדרישות, עדיין תוכל להשתמש במסלול עבור כל צומת, אך עם כמה שינויים:

- במקום בקודי צומת, המסלול יבנה מערכים המייצגים מיקום (מספר שורה) בין צמתים, בהתבסס על סדר מבוקש (לדוגמה, empname או salary).
- במקום להשתמש במחרוזת תווים בעלת אורכים שונים לכל רמה במסלול, השתמש במחרוזת בינארית בעלת אורך קבוע לכל רמה.
- ברגע שהמסלולים הבינאריים בנויים, חשב ערכי מספר שלם המייצגים סדר מסלול (מספרי שורה) והשתמש בסופו של דבר באלו כדי למיין את ההיררכיה.

האלגוריתם המרכזי לסריקת תת–העץ נשמר. השינוי הוא רק שהמסלולים נבנים בצורה שונה, ועליך למצוא דרך לחישוב מספרי השורה. ב– SQL Server 2000, כדי לחשב מספרי שורה בהתבסס על סדר מבוקש, אתה יכול להכניס את השורות לתוך טבלה עם טור identity על ידי שימוש ב– INSERT…SELECT…ORDER BY. (ראה מאמר 273586 בכתובת: http://support.microsoft.com/default.aspx?scid=kb;en-us;273586).

ב– SQL Server 2005, באפשרותך להשתמש בפונקציה ROW_NUMBER, הפשוטה ומהירה הרבה יותר מאשר החלופה ב– SQL Server 2000.

הרץ את הקוד בקטע–קוד 16–9 כדי ליצור את הפרוצדורה המאוחסנת usp_sortsubs, התואמת ל– SQL Server 2000, המיישמת לוגיקה זו.

קטע-קוד 16-9: קוד יצירה של הפרוצדורה usp_sortsubs

```
-----------------------------------------------------------------------
-- Stored Procedure: usp_sortsubs,
--   Descendants with optional level limit and sort values
--
-- Input   : @root      INT: Manager id
--           @maxlevels INT: Max number of levels to return
--           @orderby   sysname: determines sort order
--
-- Output  : Rowset: id, level and sort values
--                   of subordinates of input manager
--                   in all levels <= @maxlevels
--
-- Process :* Use a loop to load the desired subtree into #SubsPath
--          * For each node, construct a binary sort path
--          * The row number represents the node's position among
--            its siblings based on the input ORDER BY list
--          * Insert the contents of #SubPath into #SubsSort sorted
--            by the binary sortpath
--          * IDENTITY values representing the global sort value
--            in the subtree will be generated in the target
--            #SubsSort table
--          * Return all rows from #SubsSort sorted by the
--            sort value
-----------------------------------------------------------------------
 USE tempdb;
 GO
 IF OBJECT_ID('dbo.usp_sortsubs') IS NOT NULL
   DROP PROC dbo.usp_sortsubs;
 GO
 CREATE PROC dbo.usp_sortsubs
   @root      AS INT    = NULL,
   @maxlevels AS INT    = NULL,
   @orderby   AS sysnam = N'empid'
 AS

 SET NOCOUNT ON;

-- #SubsPath is a temp table that will hold binary sort paths
 CREATE TABLE #SubsPath
 (
   rownum      INT NOT NULL IDENTITY,
```

```
  nodeid   INT NOT NULL,
  lvl      INT NOT NULL,
  sortpath VARBINARY(900) NULL
);
CREATE UNIQUE CLUSTERED INDEX idx_uc_lvl_empid ON #SubsPath(lvl, nodeid);

-- #SubsPath is a temp table that will hold the final
-- integer sort values
CREATE TABLE #SubsSort
(
  nodeid   INT NOT NULL,
  lvl      INT NOT NULL,
  sortval  INT NOT NULL IDENTITY
);
CREATE UNIQUE CLUSTERED INDEX idx_uc_sortval ON #SubsSort(sortval);

-- If @root is not specified, set it to root of the tree
IF @root IS NULL
  SET @root = (SELECT empid FROM dbo.Employees WHERE mgrid IS NULL);
-- If @maxlevels is not specified, set it maximum integer
IF @maxlevels IS NULL
  SET @maxlevels = 2147483647;

DECLARE @lvl AS INT, @sql AS NVARCHAR(4000);
SET @lvl = 0;

-- Load row for input root to #SubsPath
-- The root's sort path is simply 1 converted to binary
INSERT INTO #SubsPath(nodeid, lvl, sortpath)
  SELECT empid, @lvl, CAST(1 AS BINARY(4))
  FROM dbo.Employees
  WHERE empid = @root;

-- Form a loop to load the next level of subordinates
-- to #SubsPath in each iteration
WHILE @@rowcount > 0 AND @lvl < @maxlevels
BEGIN
  SET @lvl = @lvl + 1;

  -- Insert next level of subordinates
  -- Initially, just copy parent's path to child
  -- Note that IDENTITY values will be generated in #SubsPath
```

```
  -- based on input order by list
  --
  -- Then update the path of the employees in the current level
  -- to their parent's path + their rownum converted to binary
  INSERT INTO #SubsPath(nodeid, lvl, sortpath)
    SELECT C.empid, @lvl, P.sortpath
    FROM #SubsPath AS P
      JOIN dbo.Employees AS C
        ON P.lvl = @lvl - 1
        AND C.mgrid = P.nodeid
    ORDER BY -- determines order of siblings
      CASE WHEN @orderby = N'empid'   THEN empid   END,
      CASE WHEN @orderby = N'empname' THEN empname END,
      CASE WHEN @orderby = N'salary'  THEN salary  END;

  UPDATE #SubsPath
    SET sortpath = sortpath + CAST(rownum AS BINARY(4))
  WHERE lvl = @lvl;
END

-- Load the rows from #SubsPath to @SubsSort sorted by the binary
-- sort path
-- The target identity values in the sortval column will represent
-- the global sort value of the nodes within the result subtree
INSERT INTO #SubsSort(nodeid, lvl)
  SELECT nodeid, lvl FROM #SubsPath ORDER BY sortpath;

-- Return for each node the id, level and sort value
SELECT nodeid AS empid, lvl, sortval FROM #SubsSort
ORDER BY sortval;
GO
```

פרמטרי הקלט @root ו–@maxlevels דומים לאלו בהם השתמשנו ברוטינות תת–העץ הקודמות בהן דנתי. בנוסף, הפרוצדורה המאוחסנת מקבלת את הפרמטר @orderby, בו אתה מציין שם טור לפיו אתה מעוניין למיין את האחים. הפרוצדורה המאוחסנת משתמשת בסדרה של ביטויי CASE כדי לקבוע לפי אילו ערכי טור למיין. הפרוצדורה המאוחסנת מחזירה סט תוצאה עם קודי הצומת בתת–העץ המבוקש, בנוסף לרמה ומספר שלם שהוא ערך המיון לכל צומת.

הפרוצדורה המאוחסנת סורקת את תת–העץ באופן דומה לזה של היישום האיטרטיבי הקודם בו דנתי – כלומר, רמה בכל פעם.

ראשית, עובד השורש נטען לטבלה הזמנית #SubsPath. אז, בכל איטרציה של הלולאה, הרמה הבאה של עובדים מתווספת ל–#SubsPath.

לטבלת #SubsPath יש טור identity (rownum) שייצג את מיקומו של עובד בין אחים בהתבסס על הסדר הרצוי (חלק ה– ORDER BY של המשפט INSERT SELECT). מסלול השורש מחושב כ–1 מומר ל–BINARY(4). לכל רמה של עובדים המתווספת לטבלה #SubsPath, מסלול ההורה מועתק למסלול הבן, ואז משפט UPDATE משרשר למסלול הבן את ערך ה–rownum מומר ל–BINARY(4).

בסיום הלולאה, #SubsPath מכילה את מסלול המיון הבינארי השלם עבור כל צומת.

תהליך זה יובן אולי טוב יותר על ידי דוגמה. נאמר שאתה מעוניין בתת–העץ של עובד 1 (David) ללא הגבלת רמות, ובמיון אחים לפי empname. טבלה 9–20 מציגה את ערכי ה–identity המיוצרים עבור העובדים בכל רמה.

טבלה 9–20: ערכי Identity המיוצרים עבור עובדים בכל רמה

Level 0	*Level 1*	*Level 2*	*Level 3*	*Level 4*
1 – David	2 – Eitan 3 – Ina	4 – Aaron 5 – Jiru 6 – Seraph 7 – Steve	8 – Gabriel 9 – Lilach 10 – Rita 11 – Sean	12 – Didi 13 – Emilia 14 – Michael

טבלה 9–21 מציגה את מסלולי המיון הבינאריים שנבנו לכל עובד, הבנויים מערכי המיקום של האבות המובילים לצומת.

טבלה 9–21: מסלולי מיון בינאריים שנבנו לכל עובד

lvl	*manager*	*employee*	*sortpath*
0	NULL	David (1)	1
1	David	Eitan (2)	1 2
1	David	Ina (3)	1 3
2	Eitan	Jiru (5)	1 2 5
2	Eitan	Seraph (6)	1 2 6
2	Eitan	Steve (7)	1 2 7
2	Ina	Aaron (4)	1 3 4
3	Jiru	Lilach (9)	1 2 5 9
3	Jiru	Sean (11)	1 2 5 11

lvl	*manager*	*employee*	*sortpath*				
3	Aaron	Gabriel (8)	1	3	4	8	
3	Aaron	Rita (10)	1	3	4	10	
4	Rita	Didi (12)	1	3	4	10	12
4	Rita	Emilia (13)	1	3	4	10	13
4	Rita	Michael (14)	1	3	4	10	14

הצעד הבא בפרוצדורה המאוחסנת הוא להכניס את התוכן של #SubsPath לתוך #SubsSort לפי סדר sortpath. גם ל–#SubsSort יש טור identity (sortval), שמייצג את ערכי המיון הסופיים של העובדים. טבלה 22–9 תעזור לך לראות כיצד ערכי המיון מחושבים ב–#SubsSort בהתבסס על סדר sortpath.

טבלה 22–9: ערכי מיון שלמים מחושבים בהתבסס על סדר sortpath

sortval	*lvl*	*manager*	*employee*	*sortpath*				
1	0	NULL	David (1)	1				
2	1	David	Eitan (2)	1	2			
3	2	Eitan	Jiru (5)	1	2	5		
4	3	Jiru	Lilach (9)	1	2	5	9	
5	3	Jiru	Sean (11)	1	2	5	11	
6	2	Eitan	Seraph (6)	1	2	6		
7	2	Eitan	Steve (7)	1	2	7		
8	1	David	Ina (3)	1	3			
9	2	Ina	Aaron (4)	1	3	4		
10	3	Aaron	Gabriel (8)	1	3	4	8	
11	3	Aaron	Rita (10)	1	3	4	10	
12	4	Rita	Didi (12)	1	3	4	10	12
13	4	Rita	Emilia (13)	1	3	4	10	13
14	4	Rita	Michael (14)	1	3	4	10	14

לבסוף, הפרוצדורה המאוחסנת מחזירה לכל צומת את קוד הצומת, הרמה, וערך מיון כמספר שלם. כדי לבחון את הפרוצדורה, הרץ את הקוד הבא, המציין empname כטור המיון. הקוד מפיק את הפלט המוצג בטבלה 23–9.

```
EXEC dbo.usp_sortsubs @orderby = N'empname';
```

טבלה 9–23: כל קודי העובד עם ערכי מיון המבוססים על empname

empid	*lvl*	*sortval*
1	0	1
2	1	2
5	2	3
8	3	4
10	3	5
4	2	6
6	2	7
3	1	8
7	2	9
11	3	10
9	3	11
14	4	12
12	4	13
13	4	14

כדי לקבל שלוש רמות של כפופים תחת עובד 1 ומיון אחים לפי empname, הרץ את הקוד הבא, המפיק את הפלט המוצג בטבלה 9–24:

```
EXEC dbo.usp_sortsubs
    @root = 1,
    @maxlevels = 3,
    @orderby = N'empname';
```

טבלה 9–24: תת–עץ עם הגבלת רמות, ומיון בהתבסס על empname

empid	*lvl*	*sortval*
1	0	1
2	1	2
5	2	3
8	3	4
10	3	5
4	2	6

empid	*lvl*	*sortval*
6	2	7
3	1	8
7	2	9
11	3	10
9	3	11

כדי להחזיר מאפיינים נוספים מלבד קוד עובד (למשל, שם העובד), עליך ראשית לייצר את סט התוצאה של הפרוצדורה המאוחסנת, ואז לבצע עליו join עם טבלת Employees. לדוגמה, הקוד בקטע–קוד 17–9 מחזיר את כל העובדים, כשהאחים ממוינים לפי empname, עם אינדנטציה, ומפיק את הפלט המוצג בטבלה 25–9:

קטע–קוד 17–9: קוד המחזיר את כל העובדים, כשהאחים ממוינים לפי empname

```
CREATE TABLE #Subs
(
  empid   INT NULL,
  lvl     INT NULL,
  sortval INT NULL
);
CREATE UNIQUE CLUSTERED INDEX idx_uc_sortval ON #Subs(sortval);

-- By empname
INSERT INTO #Subs(empid, lvl, sortval)
  EXEC dbo.usp_sortsubs
    @orderby = N'empname';

SELECT E.empid, REPLICATE(' | ', lvl) + E.empname AS empname
FROM #Subs AS S
  JOIN dbo.Employees AS E
    ON S.empid = E.empid
ORDER BY sortval;
```

טבלה 25–9: כל העובדים עם מיון אחים לפי empname

empid	*empname*
1	David
2	| Eitan
5	| | Jiru
8	| | | Lilach
10	| | | Sean
4	| | Seraph
6	| | Steve
3	| Ina
7	| | Aaron
11	| | | Gabriel
9	| | | Rita
14	| | | | Didi
12	| | | | Emilia
13	| | | | Michael

באופן דומה, הקוד בקטע–קוד 18–9 מחזיר את כל העובדים, כשהאחים ממוינים לפי salary, עם אינדנטציה, ומפיק את הפלט המוצג בטבלה 26–9:

קטע–קוד 18–9: קוד המחזיר את כל העובדים, עם אחים ממוינים לפי salary

```
TRUNCATE TABLE #Subs;

INSERT INTO #Subs(empid, lvl, sortval)
  EXEC dbo.usp_sortsubs
    @orderby = N'salary';

SELECT E.empid, salary, REPLICATE(' | ', lvl) + E.empname AS empname
FROM #Subs AS S
  JOIN dbo.Employees AS E
    ON S.empid = E.empid
ORDER BY sortval;
```

טבלה 26–9: כל העובדים עם אחים ממוינים לפי salary

empid	*salary*	*empname*
1	10000.00	David
2	7000.00	| Eitan
6	4500.00	| | Steve
4	5000.00	| | Seraph
5	5500.00	| | Jiru
10	3000.00	| | | Sean
8	3500.00	| | | Lilach
3	7500.00	| Ina
7	5000.00	| | Aaron
9	3000.00	| | | Rita
14	1500.00	| | | | Didi
12	2000.00	| | | | Emilia
13	2000.00	| | | | Michael
11	3000.00	| | | Gabriel

ודא שאתה מסיר את הטבלה הזמנית #Subs כאשר אתה מסיים:

```
DROP TABLE #Subs;
```

היישום של אלגוריתם דומה ב– SQL Server 2005 פשוט ומהיר יותר משמעותית, בעיקר מכיוון שהוא משתמש ב–CTEs ובפונקציה ROW_NUMBER.

הרץ את הקוד בקטע–קוד 19–9 כדי להחזיר את תת–העץ של עובד 1, עם אחים ממוינים לפי empname עם אינדנטציה (indentation), וכדי לייצר את הפלט המוצג בטבלה 27–9.

קטע–קוד 19–9: החזרת כל העובדים בהיררכיה עם אחים ממוינים לפי empname, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    -- Path of root is 1 (binary)
```

```
    CAST(1 AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    -- Path of child = parent's path + child row number (binary)
    P.sortpath + CAST(
      ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        ORDER BY C.empname) -- sort col(s)
      AS BINARY(4))
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval,
  REPLICATE(' | ', lvl) + empname AS empname
FROM SubsCTE
ORDER BY sortval;
```

טבלה 9–27: עובדים עם מיון המתבסס על empname, פתרון CTE

empid	*sortval*	*empname*
1	1	David
2	2	| Eitan
5	3	| | Jiru
8	4	| | | Lilach
10	5	| | | Sean
4	6	| | Seraph
6	7	| | Steve
3	8	| Ina
7	9	| | Aaron
11	10	| | | Gabriel
9	11	| | | Rita
14	12	| | | | Didi
12	13	| | | | Emilia
13	14	| | | | Michael

שאילתת איבר העוגן מחזירה את השורש, עם 1 כמסלול הבינארי. שאילתת האיבר הרקורסיבי מחשבת את מספר השורה של עובד בין אחים בהתבסס על סדר empname, ומשרשרת למסלול ההורה את מספר השורה מומר ל–BINARY(4).

השאילתה החיצונית פשוט מחשבת מספרי שורה כדי לייצר את ערכי המיון, בהתבסס על סדר מסלול בינארי, והיא ממיינת את תת–העץ לפי ערכי מיון אלו, כשהיא מוסיפה אינדנטציה בהתבסס על הרמה המחושבת.

אם אתה מעוניין שאחים ימוינו בדרך אחרת, עליך לשנות רק את רשימת ה– ORDER BY של הפונקציה ROW_NUMBER בשאילתת האיבר הרקורסיבי. בקטע–קוד 20–9 יש את השינויים הממיינים אחים לפי salary, ומייצרים את הפלט המוצג בטבלה 28–9.

קטע–קוד 20–9: החזרת כל העובדים בהיררכיה עם אחים ממוינים לפי salary, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, salary, 0 AS lvl,
    -- Path of root = 1 (binary)
    CAST(1 AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, C.salary, P.lvl + 1,
    -- Path of child = parent's path + child row number (binary)
    P.sortpath + CAST(
      ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        ORDER BY C.salary) -- sort col(s)
      AS BINARY(4))
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, salary, ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval,
  REPLICATE(' | ', lvl) + empname AS empname
FROM SubsCTE
ORDER BY sortval;
```

טבלה 28–9: עובדים עם מיון המבוסס על salary, פתרון CTE

empid	*salary*	*sortval*	*empname*
1	10000.00	1	David
2	7000.00	2	| Eitan
6	4500.00	3	| | Steve
4	5000.00	4	| | Seraph
5	5500.00	5	| | Jiru
10	3000.00	6	| | | Sean
8	3500.00	7	| | | Lilach
3	7500.00	8	| Ina
7	5000.00	9	| | Aaron
9	3000.00	10	| | | Rita
14	1500.00	11	| | | | Didi
12	2000.00	12	| | | | Emilia
13	2000.00	13	| | | | Michael
11	3000.00	14	| | | Gabriel

שים לב: אם עליך למיין אחים לפי טור מיון יחיד מטיפוס INT (לדוגמה, לפי empid), אתה יכול לבנות את מסלול המיון הבינארי מערכי טור המיון עצמם במקום ממספרי שורה המבוססים על טור זה.

מחזוריות

מחזוריות (cycles) בגרפים היא קיום מסלולים המתחילים ומסתיימים באותו צומת. ישנם תרחישים בהם מחזוריות היא טבעית (למשל, במערכות כבישים). אם יש לך מחזוריות במה שאמור להיות גרף לא–מעגלי, דבר זה עשוי להעיד על בעיה בנתונים שלך. בכל מקרה, עליך להיות מסוגל לזהות מעגלים. אם מחזוריות מעידה על בעיה בנתונים, עליך לזהות את הבעיה ולתקן אותה. אם מחזוריות היא טבעית, כאשר אתה סורק את הגרף אינך מעוניין לחזור שוב ושוב לאותה נקודה.

זיהוי מחזוריות עם T-SQL עשוי להיות משימה מאוד מורכבת ויקרה. אף על פי כן, אראה לך כיצד לזהות מחזוריות על ידי שיטה פשוטה למדי בעלת ביצועים סבירים, תוך הסתמכות על מסלול אבות בו דנתי קודם. למטרות הדגמה, אשתמש בשיטה זו כדי לזהות מחזוריות בעץ המיוצג על ידי טבלת Employees, אך תוכל להפעיל שיטה זו גם על יערות וכן על גרפים כללים יותר, כפי שאדגים בהמשך.

נניח שדידי (Didi – קוד עובד 14) אינה מרוצה ממיקומה בהיררכיית הניהול של החברה. במקרה, דידי היא גם מנהלת מסד הנתונים ויש לה גישה מלאה לטבלת Employees. דידי מריצה את הקוד הבא, ההופך אותה למנהלת של המנכ"ל ומציג מחזוריות:

```
UPDATE dbo.Employees SET mgrid = 14 WHERE empeid = 1;
```

טבלת Employees כרגע מכילה את המעגל הבא של קודי עובד: 1→14→9→7→3→1.

כנקודת פתיחה, אשתמש באחד הפתרונות שהצגתי קודם, הבונה מסלול אבות. בדוגמאות שלי, אשתמש בפתרון CTE, אך ניתן ליישם כמובן את אותה לוגיקה בפתרון ה–UDF ב– SQL Server 2000.

בתיאור פשוט, מחזוריות מזוהה כאשר אתה עוקב אחר מסלול המוביל לצומת נתון, אם מסלול ההורה שלו מכיל כבר את קוד הצומת של הבן. באפשרותך לעקוב אחר המעגלים על ידי תחזוקת טור cycle, שיכיל 0 אם לא זוהה כל מעגל ו–1 אם זוהה מעגל. באיבר העוגן של פתרון ה–CTE, ערך הטור cycle הוא הקבוע 0, מכיוון שאין ספק שלא קיימת מחזוריות ברמת השורש. בשאילתת האיבר הרקורסיבי, השתמש בביטוי LIKE כדי לבדוק האם מסלול הורה מכיל את קוד הצומת של בן. החזר 1 אם הוא מכיל ו–0 אחרת. שים לב לחשיבות של הנקודות בהתחלה ובסיום הן של המסלול והן של התבנית – ללא הנקודות, תקבל התאמה בלתי רצויה של קוד עובד n (לדוגמה n = 3), אם המסלול מכיל קוד עובד nm (לדוגמה m = 15, nm = 315). קטע–קוד 21–9 מציג את הקוד המחזיר תת–עץ עם חישוב מסלול אבות ויש לו את התוספת של חישוב הטור cycle. אם תריץ את הקוד בקטע–קוד 21–9, הוא תמיד יעוף לאחר 100 רמות (ערך ברירת המחדל של MAXRECURSION), מכיוון שמחזוריות מזוהה אך לא נמנעת.

קטע–קוד 21–9: זיהוי מחזוריות, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Obviously root has no cycle
    0 AS cycle
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL
```

```
  SELECT C.empid, C.empname, P.lvl + 1,
    CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Cycle detected if parent's path contains child's id
    CASE WHEN P.path LIKE '%.' + CAST(C.empid AS VARCHAR(10)) + '.%'
      THEN 1 ELSE 0 END
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, empname, cycle, path
FROM SubsCTE;
```

עליך להימנע ממחזוריות, או במילים אחרות, לא להמשיך במסלול שבו זוהו מעגלים. כדי להשיג זאת, הוסף מסנן לאיבר הרקורסיבי המחזיר בן רק אם ערך ה–cycle של ההורה שלו הוא 0. הקוד בקטע–קוד 22–9 מכיל את לוגיקת ההימנעות ממחזוריות זו, ומפיק את הפלט המוצג בטבלה 29–9.

קטע–קוד 22–9: תת–עץ ללא סריקה מחזורית של מעגלים, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Obviously root has no cycle
    0 AS cycle
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
    CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
         AS VARCHAR(MAX)) AS path,
    -- Cycle detected if parent's path contains child's id
```

```
    CASE WHEN P.path LIKE '%.' + CAST(C.empid AS VARCHAR(10)) + '.%'
      THEN 1 ELSE 0 END
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
      AND P.cycle = 0 -- do not pursue branch for parent with cycle
)
SELECT empid, empname, cycle, path
FROM SubsCTE;
```

טבלה 9–29: Employees ללא סריקה מחזורית של מעגלים

empid	*empname*	*cycle*	*path*
1	David	0	.1.
2	Eitan	0	.1.2.
3	Ina	0	.1.3.
7	Aaron	0	.1.3.7.
11	Gabriel	0	.1.3.7.11.
9	Rita	0	.1.3.7.9.
12	Emilia	0	.1.3.7.9.12.
13	Michael	0	.1.3.7.9.13.
14	Didi	0	.1.3.7.9.14.
1	**David**	**1**	**.1.3.7.9.14.1.**
4	Seraph	0	.1.2.4.
5	Jiru	0	.1.2.5.
6	Steve	0	.1.2.6.
10	Sean	0	.1.2.5.10.
8	Lilach	0	.1.2.5.8.

שים לב בפלט, שבפעם השנייה שהקוד הגיע לעובד 1, זוהה עבורו מעגל, והקוד לא המשיך לסרוק בצורה מחזורית את המסלול בגרף המעגלי. זוהי לרוב כל הלוגיקה שעליך להוסיף. במקרה שלנו, המחזוריות מעידה על בעיה בנתונים אותה יש לתקן. כדי לבודד רק את המסלול המחזורי (במקרה שלנו, מימין לשמאל .1.3.7.9.14.1.), הוסף את המסנן cycle = 1 לשאילתה החיצונית כפי שמוצג בקטע–קוד 9–23.

קטע–קוד 23–9: בידוד מסלולים מחזוריים, פתרון CTE

```
DECLARE @root AS INT;
SET @root = 1;

WITH SubsCTE
AS
(
  SELECT empid, empname, 0 AS lvl,
     CAST('.' + CAST(empid AS VARCHAR(10)) + '.'
          AS VARCHAR(MAX)) AS path,
     -- Obviously root has no cycle
     0 AS cycle
  FROM dbo.Employees
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, C.empname, P.lvl + 1,
     CAST(P.path + CAST(C.empid AS VARCHAR(10)) + '.'
          AS VARCHAR(MAX)) AS path,
     -- Cycle detected if parent's path contains child's id
     CASE WHEN P.path LIKE '%.' + CAST(C.empid AS VARCHAR(10)) + '.%'
       THEN 1 ELSE 0 END
  FROM SubsCTE AS P
    JOIN dbo.Employees AS C
      ON C.mgrid = P.empid
      AND P.cycle = 0
)
SELECT path FROM SubsCTE WHERE cycle = 1;
```

כעת כשזוהה המסלול המחזורי, תוכל לתקן את הנתונים על ידי הרצת הקוד הבא:

```
UPDATE dbo.Employees SET mgrid = NULL WHERE empid = 1;
```

סביר להניח שדידי תמצא את עצמה מובטלת.

מסלול אבות ממומש

עד כה הצגתי פתרונות בהם מסלולים חושבו כאשר הופעל הקוד. בפתרון מסלול האבות הממומש (materialized path), המסלולים יאוחסנו כך שלא יהיה צורך לחזור ולחשב אותם שוב ושוב. אתה למעשה מאחסן מסלול אבות ורמה לכל צומת בעץ בשני טורים נוספים. הפתרון יכול להיות מופעל על עצים בלבד – ואפשרי גם על יערות.

קיימים שני יתרונות עיקריים לגישה זו על פני הגישה האיטרטיבית/רקורסיבית. השאילתות הן פשוטות יותר ומבוססות–סטים (ללא הסתמכות על CTEs רקורסיביים). כמו כן, לשאילתות לרוב ביצועים מהירים יותר, שכן הן יכולות להישען על אינדקסים שהוגדרו על הטור המכיל את המסלול.

עם זאת, כעת כשיש לך שני טורים נוספים בטבלה, עליך לשמור אותם מסונכרנים עם העץ כשהוא עובר שינויים. עלות השינויים תקבע האם סביר לסנכרן את המסלול וערכי הרמה בכל שינוי של העץ. לדוגמה, מהי ההשפעה של הוספת עלה חדש לעץ? אני אוהב להתייחס להשפעה של שינוי כזה באופן לא–פורמלי כ"אפקט הזַעֲזוּע". למזלנו, כפי שאתאר מייד, הוספת עלים חדשים גורמת לאפקט זעזוע שולי. כמו כן, ההשפעה של הסרה או הזזה של תת–עץ קטן היא לרוב אינה משמעותית.

מסלול האבות עשוי להיות ארוך כאשר העץ עמוק – במילים אחרות, כאשר קיימות רמות רבות של מנהלים. SQL Server מגביל את גודל מפתחות האינדקס ל–900 בתים. כדי להשיג את יתרונות הביצועים של אינדקס על טור המסלול, תצטרך להגביל אותו ל–900 בתים. בטרם תהיה מודאג מעובדה זו, נסה לחשוב במונחים מעשיים: 900 בתים יכולים להכיל עצים בעלי מאות רמות. האם העץ שלך יגיע אי פעם ליותר ממאות רמות? אני מודה שמעולם לא נדרשתי לעצב היררכיה בעלת מאות רמות. בקיצור, הפעל הגיון פשוט וחשוב במונחים מעשיים.

תחזוקת נתונים

ראשית הרץ את הקוד בקטע–קוד 24–9 כדי ליצור את טבלת Employees עם הטורים החדשים lvl ו–path.

קטע–קוד 24–9: מבנה טבלת Employees עם מסלולי אבות ממומשים

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Employees') IS NOT NULL
  DROP TABLE dbo.Employees;
GO
CREATE TABLE dbo.Employees
(
  empid   INT          NOT NULL PRIMARY KEY NONCLUSTERED,
  mgrid   INT          NULL     REFERENCES dbo.Employees,
  empname VARCHAR(25)  NOT NULL,
  salary  MONEY        NOT NULL,
  lvl     INT          NOT NULL,
  path    VARCHAR(900) NOT NULL UNIQUE CLUSTERED
);
CREATE UNIQUE INDEX idx_unc_mgrid_empid ON dbo.Employees(mgrid, empid);
GO
```

כדי לטפל בשינויים בעץ, מומלץ להשתמש בפרוצדורות מאוחסנות שיטפלו גם בערכי lvl ו–path. לחלופין, תוכל להשתמש בטריגרים, והלוגיקה שלהם תהיה דומה מאוד לזו שבפרוצדורות המאוחסנות להלן.

הוספת עובדים שאינם מנהלים (עלים)

הבה נתחיל עם טיפול ב–inserts. הלוגיקה של פרוצדורת ה–inserts היא פשוטה. אם העובד החדש הוא עובד שורש (כלומר, קוד המנהל הוא NULL), הרמה שלו היא 0 והמסלול שלו הוא '.' + employee id + '.'. כפי שתוכל לנחש, אפקט הזעזוע כאן הוא שולי. אין צורך לערוך כל שינויים לעובדים אחרים, וכדי לחשב את ערכי lvl ו–path של העובד החדש, עליך רק לבצע שאילתה על ההורה של העובד.

הרץ את הקוד בקטע–קוד 25–9, כדי ליצור את הפרוצדורה המאוחסנת usp_insertemp, והרץ את הקוד בקטע–קוד 26–9 כדי למלא את טבלת Employees בנתונים לדוגמה.

קטע–קוד 25–9: קוד יצירה של הפרוצדורה usp_insertemp

```
-----------------------------------------------------------------
-- Stored Procedure: usp_insertemp,
--   Inserts new employee who manages no one into the table
-----------------------------------------------------------------
 USE tempdb;
 GO
 IF OBJECT_ID('dbo.usp_insertemp') IS NOT NULL
   DROP PROC dbo.usp_insertemp;
 GO
 CREATE PROC dbo.usp_insertemp
   @empid   INT,
   @mgrid   INT,
   @empname VARCHAR(25),
   @salary  MONEY
 AS

 SET NOCOUNT ON;

-- Handle case where the new employee has no manager (root)
 IF @mgrid IS NULL
   INSERT INTO dbo.Employees(empid, mgrid, empname, salary, lvl, path)
     VALUES(@empid, @mgrid, @empname, @salary,
       0, '.' + CAST(@empid AS VARCHAR(10)) + '.');
-- Handle subordinate case (non-root)
```

```
ELSE
  INSERT INTO dbo.Employees(empid, mgrid, empname, salary, lvl, path)
    SELECT @empid, @mgrid, @empname, @salary,
      lvl + 1, path + CAST(@empid AS VARCHAR(10)) + '.'
    FROM dbo.Employees
    WHERE empid = @mgrid;
GO
```

קטע-קוד 9-26: נתונים לדוגמה עבור עובדים עם מסלול

```
EXEC dbo.usp_insertemp
  @empid = 1, @mgrid = NULL, @empname = 'David', @salary = $10000.00;
EXEC dbo.usp_insertemp
  @empid = 2, @mgrid = 1, @empname = 'Eitan', @salary = $7000.00;
EXEC dbo.usp_insertemp
  @empid = 3, @mgrid = 1, @empname = 'Ina', @salary = $7500.00;
EXEC dbo.usp_insertemp
  @empid = 4, @mgrid = 2, @empname = 'Seraph', @salary = $5000.00;
EXEC dbo.usp_insertemp
  @empid = 5, @mgrid = 2, @empname = 'Jiru', @salary = $5500.00;
EXEC dbo.usp_insertemp
  @empid = 6, @mgrid = 2, @empname = 'Steve', @salary = $4500.00;
EXEC dbo.usp_insertemp
  @empid = 7, @mgrid = 3, @empname = 'Aaron', @salary = $5000.00;
EXEC dbo.usp_insertemp
  @empid = 8, @mgrid = 5, @empname = 'Lilach', @salary = $3500.00;
EXEC dbo.usp_insertemp
  @empid = 9, @mgrid = 7, @empname = 'Rita', @salary = $3000.00;
EXEC dbo.usp_insertemp
  @empid = 10, @mgrid = 5, @empname = 'Sean', @salary = $3000.00;
EXEC dbo.usp_insertemp
  @empid = 11, @mgrid = 7, @empname = 'Gabriel', @salary = $3000.00;
EXEC dbo.usp_insertemp
  @empid = 12, @mgrid = 9, @empname = 'Emilia', @salary = $2000.00;
EXEC dbo.usp_insertemp
  @empid = 13, @mgrid = 9, @empname = 'Michael', @salary = $2000.00;
EXEC dbo.usp_insertemp
  @empid = 14, @mgrid = 9, @empname = 'Didi', @salary = $1500.00;
```

הרץ את השאילתה הבאה כדי לבדוק את התוכן של טבלת Employees כולל מסלולים כפי שמוצג בטבלה 30–9:

```
SELECT empid, mgrid, empname, salary, lvl, path
FROM dbo.Employees
ORDER BY path;
```

טבלה 30–9: עובדים עם מסלולי אבות ממומשים

empid	*mgrid*	*empname*	*salary*	*lvl*	*path*
1	NULL	David	10000.0000	0	.1.
2	1	Eitan	7000.0000	1	.1.2.
4	2	Seraph	5000.0000	2	.1.2.4.
5	2	Jiru	5500.0000	2	.1.2.5.
10	5	Sean	3000.0000	3	.1.2.5.10.
8	5	Lilach	3500.0000	3	.1.2.5.8.
6	2	Steve	4500.0000	2	.1.2.6.
3	1	Ina	7500.0000	1	.1.3.
7	3	Aaron	5000.0000	2	.1.3.7.
11	7	Gabriel	3000.0000	3	.1.3.7.11.
9	7	Rita	3000.0000	3	.1.3.7.9.
12	9	Emilia	2000.0000	4	.1.3.7.9.12.
13	9	Michael	2000.0000	4	.1.3.7.9.13.
14	9	Didi	1500.0000	4	.1.3.7.9.14.

הזזת תת–עץ

הזזת תת–עץ היא מעט ערמומית. שינוי במנהל של מישהו, משפיע על השורה של עובד זה ושל כל הכפופים לו. הקלטים הם קוד עובד השורש של תת–העץ וקוד ההורה החדש (מנהל) של עובד שורש זה. ערכי הרמה והמסלול של כל העובדים בתת–העץ עומדים להיות מושפעים, כך שעליך להיות מסוגל לבודד את תת–עץ זה וגם לגלות כיצד לשנות את ערכי הרמה והמסלול של כל החברים בתת–העץ. כדי לבודד את תת–העץ, אתה מבצע join בין שורת השורש (R) לבין טבלת Employees (E) בהתבסס על E.path LIKE R.path + '%'. כדי לחשב את השינויים ברמה ובמסלול, אתה צריך גישה לשורות הן של המנהל הישן של השורש (OM) והן של המנהל החדש (NM). ערך הרמה החדש לכל הצמתים הוא הרמה הנוכחית שלהם ועוד ההפרש ברמות בין רמת המנהל החדש לרמת המנהל הישן. לדוגמה, אם אתה מזיז תת–עץ למיקום חדש, כך שההבדל ברמות בין המנהל

החדש למנהל הישן הוא 2, עליך להוסיף 2 לערך הרמה של כל העובדים בתת–העץ המושפע. בדומה, כדי לתקן את ערך המסלול של כל הצמתים בתת–העץ, עליך להסיר את התחילית המכילה את המסלול של מנהל השורש הישן ולהחליף אותה במסלול המנהל החדש. דבר זה ניתן לבצע על ידי שימוש בפונקציה STUFF.

הרץ את הקוד בקטע–קוד 27–9 כדי ליצור את הפרוצדורה המאוחסנת usp_movesubtree, המיישמת את הלוגיקה שזה עתה הסברתי.

קטע–קוד 27–9: קוד יצירה של הפרוצדורה usp_movesubtree

```
-----------------------------------------------------------------------
-- Stored Procedure: usp_movesubtree,
--   Moves a whole subtree of a given root to a new location
--   under a given manager
-----------------------------------------------------------------------
 USE tempdb;
 GO
 IF OBJECT_ID('dbo.usp_movesubtree') IS NOT NULL
   DROP PROC dbo.usp_movesubtree;
 GO
 CREATE PROC dbo.usp_movesubtree
   @root  INT,
   @mgrid INT
 AS

 SET NOCOUNT ON;

 BEGIN TRAN;
   -- Update level and path of all employees in the subtree (E)
   -- Set level =
   --   current level + new manager's level - old manager's level
   -- Set path =
   --   in current path remove old manager's path
   --   and substitute with new manager's path
   UPDATE E
     SET lvl  = E.lvl + NM.lvl - OM.lvl,
         path = STUFF(E.path, 1, LEN(OM.path), NM.path)
   FROM dbo.Employees AS E          -- E = Employees    (subtree)
     JOIN dbo.Employees AS R        -- R = Root         (one row)
       ON R.empid = @root
       AND E.path LIKE R.path + '%'
     JOIN dbo.Employees AS OM       -- OM = Old Manager (one row)
       ON OM.empid = R.mgrid
     JOIN dbo.Employees AS NM       -- NM = New Manager (one row)
        ON NM.empid = @mgrid;
```

```
  -- Update root's new manager
  UPDATE dbo.Employees SET mgrid = @mgrid WHERE empid = @root;
COMMIT TRAN;
GO
```

יישום פרוצדורה מאוחסנת זו הוא פשטני והוא מסופק למטרות הדגמה. התנהגות טובה אינה מובטחת כאשר המשתמש מספק פרמטרים שאינם תקניים. בשביל להפוך את התהליך הזה לרציני יותר, עליך לבדוק גם את הקלטים כדי לוודא שנדחים ניסיונות להפוך מישהו למנהל של עצמו או לייצר מחזוריות. לדוגמה, דבר זה ניתן להשגה על ידי שימוש בביטוי EXISTS עם משפט SELECT, אשר מייצר סט תוצאה עם המסלולים החדשים, ובדיקה שקודי העובד לא מופיעים במסלול של המנהל שלהם.

כדי לבדוק את הפרוצדורה, בחן את העץ המופיע בטבלה 31–9 לפני הזזת תת–העץ:

```
SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
FROM dbo.Employees
ORDER BY path;
```

טבלה 31–9: עובדים לפני הזזת תת–עץ

empid	*empname*	*lvl*	*path*
1	David	0	.1.
2	| Eitan	1	.1.2.
4	| | Seraph	2	.1.2.4.
5	| | Jiru	2	.1.2.5.
10	| | | Sean	3	.1.2.5.10.
8	| | | Lilach	3	.1.2.5.8.
6	| | Steve	2	.1.2.6.
3	| Ina	1	.1.3.
7	| | Aaron	2	.1.3.7.
11	| | | Gabriel	3	.1.3.7.11.
9	| | | Rita	3	.1.3.7.9.
12	| | | | Emilia	4	.1.3.7.9.12.
13	| | | | Michael	4	.1.3.7.9.13.
14	| | | | Didi	4	.1.3.7.9.14.

כעת הרץ את הקוד הבא כדי להזיז את תת–העץ של Aaron תחת Sean, ובחן את עץ התוצאה המוצג בטבלה 32–9 כדי לוודא שתת–העץ הוזז בצורה נכונה:

```
BEGIN TRAN;

  EXEC dbo.usp_movesubtree
  @root  = 7,
  @mgrid = 10;

  -- After moving subtree
  SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
  FROM dbo.Employees
  ORDER BY path;

ROLLBACK TRAN; -- rollback used in order not to apply the change
```

שים לב: לשינוי מבוצע rollback לצורך הדגמה בלבד, כדי שהנתונים יהיו זהים בתחילת כל קוד בדיקה.

טבלה 32–9: עובדים לאחר הזזת תת–עץ

empid	*empname*	*lvl*	*path*
1	David	0	.1.
2	| Eitan	1	.1.2.
4	| | Seraph	2	.1.2.4.
5	| | Jiru	2	.1.2.5.
10	| | | Sean	3	.1.2.5.10.
7	| | | | Aaron	4	.1.2.5.10.7.
11	| | | | | Gabriel	5	.1.2.5.10.7.11.
9	| | | | | Rita	5	.1.2.5.10.7.9.
12	| | | | | | Emilia	6	.1.2.5.10.7.9.12.
13	| | | | | | Michael	6	.1.2.5.10.7.9.13.
14	| | | | | | Didi	6	.1.2.5.10.7.9.14.
8	| | | Lilach	3	.1.2.5.8.
6	| | Steve	2	.1.2.6.
3	| Ina	1	.1.3.

הסרת תת–עץ

הסרת תת–עץ היא משימה פשוטה, אתה פשוט מוחק את כל העובדים שבערך המסלול שלהם יש את מסלול השורש של תת–העץ כתחילית. כדי לבדוק פתרון זה, בחן את המצב הנוכחי של העץ המוצג בטבלה 33–9 על ידי הרצת השאילתה הבאה:

```
SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
FROM dbo.Employees
ORDER BY path;
```

טבלה 33–9: עובדים לפני מחיקת תת–עץ

empid	*empname*	*lvl*	*path*
1	David	0	.1.
2	| Eitan	1	.1.2.
4	| | Seraph	2	.1.2.4.
5	| | Jiru	2	.1.2.5.
10	| | | Sean	3	.1.2.5.10.
8	| | | Lilach	3	.1.2.5.8.
6	| | Steve	2	.1.2.6.
3	| Ina	1	.1.3.
7	| | Aaron	2	.1.3.7.
11	| | | Gabriel	3	.1.3.7.11.
9	| | | Rita	3	.1.3.7.9.
12	| | | | Emilia	4	.1.3.7.9.12.
13	| | | | Michael	4	.1.3.7.9.13.
14	| | | | Didi	4	.1.3.7.9.14.

הפעל את הקוד הבא, אשר מסיר את Aaron והכפופים לו קודם ואז מציג את עץ התוצאה המוצג בטבלה 34–9:

```
BEGIN TRAN;

  DELETE FROM dbo.Employees
  WHERE path LIKE
    (SELECT M.path + '%'
     FROM dbo.Employees as M
     WHERE M.empid = 7);

  -- After deleting subtree
  SELECT empid, REPLICATE(' | ', lvl) + empname AS empname, lvl, path
```

```
  FROM dbo.Employees
  ORDER BY path;

ROLLBACK TRAN; -- rollback used in order not to apply the change
```

טבלה 34–9: עובדים לאחר מחיקת תת–עץ

empid	*empname*	*lvl*	*path*
1	David	0	.1.
2	| Eitan	1	.1.2.
4	| | Seraph	2	.1.2.4.
5	| | Jiru	2	.1.2.5.
10	| | | Sean	3	.1.2.5.10.
8	| | | Lilach	3	.1.2.5.8.
6	| | Steve	2	.1.2.6.
3	| Ina	1	.1.3.

ביצוע שאילתות

ביצוע שאילתות (querying) על הנתונים בפתרון מסלול האבות הממומש הוא פשוט ואלגנטי. עבור בקשות הקשורות לתת–עצים, ה–optimizer יכול תמיד להשתמש באינדקס–clustered או באינדקס–מכסה שאתה יוצר על הטור path. אם אתה יוצר אינדקס–nonclustered לא–מכסה על הטור path, ה–optimizer יוכל להשתמש בו אם השאילתה סלקטיבית מספיק.

הבה נבחן בקשות מיוחדות מעץ. לכל בקשה, אספק שאילתה לדוגמה ולאחריה הפלט שלה.

החזר את תת–העץ עם שורש נתון. תקבל את הפלט המוצג בטבלה 35–9:

```
SELECT REPLICATE(' | ', E.lvl - M.lvl) + E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3 -- root
    AND E.path LIKE M.path + '%'
ORDER BY E.path;
```

טבלה 35–9: תת–עץ עם שורש נתון

Ina
| Aaron
| | Gabriel
| | Rita
| | | Emilia
| | | Michael
| | | Didi

השאילתה מבצעת join בין שני מופעים של Employees. אחד מייצג את המנהלים (M) ומסונן לפי עובד השורש הנתון. השני מייצג את העובדים בתת–העץ (E). תת–העץ מזוהה על ידי שימוש בביטוי הלוגי הבא בתנאי ה–join : E.path LIKE M.path + '%', המזהה עובד ככפוף אם הוא מכיל את מסלול השורש כתחילית. אינדנטציה מושגת על ידי הכפלת המחרוזת (' | ') כמספר הפעמים השווה לרמת העובד בתוך תת–העץ. הפלט ממוין לפי מסלול העובד.

כדי להשמיט את שורש תת–העץ (מנהל ברמה העליונה) מהפלט, הוסף קו תחתון לפני סימן האחוז בתבנית ה–LIKE:

```
SELECT REPLICATE(' | ', E.lvl - M.lvl - 1) + E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '_%'
ORDER BY E.path;
```

תקבל את הפלט המוצג בטבלה 36–9:

טבלה 36–9: תת–עץ של שורש נתון, ללא שורש

Aaron
| Gabriel
| Rita
| | Emilia
| | Michael
| | Didi

עם תוספת הקו התחתון בתנאי ה–LIKE, עובד מוחזר רק אם המסלול שלו מתחיל עם מסלול השורש ויש לו לפחות תו עוקב אחד.

כדי להחזיר צמתי עלה תחת שורש נתון (כולל השורש עצמו אם הוא עלה), הוסף ביטוי NOT EXISTS בשביל לזהות רק עובדים שאינם מנהלים של עובד אחר:

```
SELECT E.empid, E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '%'
WHERE NOT EXISTS
  (SELECT *
   FROM dbo.Employees AS E2
   WHERE E2.mgrid = E.empid);
```

תקבל את הפלט המוצג בטבלה 9–37.

טבלה 9–37: צמתי עלה תחת שורש נתון

empid	*empname*
11	Gabriel
12	Emilia
13	Michael
14	Didi

כדי להחזיר תת–עץ, תוך הגבלת מספר הרמות תחת השורש, הוסף מסנן בתנאי ה–join המגביל את הפרש הרמה בין העובד לשורש:

```
SELECT REPLICATE(' | ', E.lvl - M.lvl) + E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '%'
    AND E.lvl - M.lvl <= 2
ORDER BY E.path;
```

תקבל את הפלט המוצג בטבלה 9–38.

טבלה 38–9: תת–עץ עם שורש נתון, הגבלת רמות

Ina
| Aaron
| | Gabriel
| | Rita

כדי להחזיר את הצמתים בדיוק n רמות תחת שורש נתון, השתמש באופרטור שוויון (=) כדי לזהות את הפרש הרמות הספציפי במקום אופרטור קטן שווה (<=):

```
SELECT E.empid, E.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON M.empid = 3
    AND E.path LIKE M.path + '%'
    AND E.lvl - M.lvl = 2;
```

טבלה 39–9: צמתים הנמצאים בדיוק n רמות תחת שורש נתון

empid	*empname*
11	Gabriel
9	Rita

כדי להחזיר שרשרת ניהול של צומת נתון, אתה משתמש בשאילתה דומה לשאילתת תת–העץ, עם הבדל קטן – אתה מסנן קוד עובד ספציפי, בניגוד לסינון קוד מנהל ספציפי:

```
SELECT REPLICATE(' | ', M.lvl) + M.empname
FROM dbo.Employees AS E
  JOIN dbo.Employees AS M
    ON E.empid = 14
    AND E.path LIKE M.path + '%'
ORDER BY E.path;
```

תקבל את הפלט המוצג בטבלה 40–9.

טבלה 40–9: שרשרת ניהול של עובד 14

David
| Ina
| | Aaron
| | | Rita
| | | | Didi

תקבל את כל המנהלים שהמסלול שלהם הוא תחילית של מסלול העובד הנתון.

שים לב שקיים הבדל חשוב בביצועים בין בקשה לתת–עץ ובקשה לאבות, למרות שהן נראות דומות מאוד. לכל שאילתה, M.path או E.path הוא קבוע. אם M.path הוא קבוע, E.path LIKE M.path + '%' משתמש באינדקס, מכיוון שהוא מבקש את כל המסלולים בעלי תחילית נתונה. שאילתת תת–העץ יכולה לבצע seek בתוך אינדקס למסלול הראשון אשר עומד במסנן, והיא יכולה לבצע סריקה ימינה עד שהיא מגיעה למסלול האחרון אשר עומד במסנן. במילים אחרות, מתבצעת גישה רק למסלולים הרלוונטיים באינדקס בעוד שבשאילתת האבות, **כל** המסלולים חייבים להיסרק כדי לבדוק אם הם עומדים במסנן. המשמעות היא ביצוע table/index scan מלא. בטבלאות גדולות, הדבר מתורגם לשאילתה איטית. כדי לטפל בבקשות לאבות בצורה יעילה יותר, אתה יכול ליצור פונקציה המקבלת קוד עובד כקלט, מפצלת את המסלול שלו, ומחזירה טבלה עם קודי הצומת של המסלול בשורות נפרדות. תוכל לבצע join בין טבלה זו לבין העץ ולהשתמש בפעולות index seek לאיתור קודי העובדים הספציפיים במסלול. פונקציית הפיצול משתמשת בטבלת עזר של מספרים, בה דנתי בפרק 4 בסעיף "טבלת עזר של מספרים". אם אין לך כרגע טבלת Nums ב–tempdb, ראשית צור אותה על ידי הרצת הקוד בקטע–קוד 28–9.

קטע–קוד 28–9: יצירה ומילוי של טבלת עזר של מספרים

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Nums') IS NOT NULL
  DROP TABLE dbo.Nums;
GO
CREATE TABLE Nums(n INT NOT NULL PRIMARY KEY);
DECLARE @max AS INT, @rc AS INT;
SET @max = 8000;
SET @rc = 1;

INSERT INTO Nums VALUES(1);
WHILE @rc * 2 <= @max
```

```
BEGIN
  INSERT INTO dbo.Nums SELECT n + @rc FROM dbo.Nums;
  SET @rc = @rc * 2;
END

INSERT INTO dbo.Nums
  SELECT n + @rc FROM dbo.Nums WHERE n + @rc <= @max;
```

הרץ את הקוד בקטע–קוד 29–9 כדי ליצור את הפונקציה fn_splitpath.

קטע–קוד 29–9: קוד יצירה של הפונקציה fn_splitpath

```
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_splitpath') IS NOT NULL
  DROP FUNCTION dbo.fn_splitpath;
GO
CREATE FUNCTION dbo.fn_splitpath(@empid AS INT) RETURNS TABLE
AS
RETURN
  SELECT
    n - LEN(REPLACE(LEFT(path, n), '.', '')) AS pos,
    CAST(SUBSTRING(path, n + 1,
           CHARINDEX('.', path, n+1) - n - 1) AS INT) AS empid
  FROM dbo.Employees
    JOIN dbo.Nums
      ON empid = @empid
      AND n < LEN(path)
      AND SUBSTRING(path, n, 1) = '.'
GO
```

תוכל למצוא פרטים על הלוגיקה שמאחורי שיטת הפיצול שהפונקציה מיישמת, בפרק 5 בסעיף "הפרדת אלמנטים".

כדי לבדוק את הפונקציה, הרץ את הקוד הבא, המפצל את המסלול של עובד 14 ומפיק את הפלט המוצג בטבלה 41–9:

```
SELECT pos, empid FROM dbo.fn_splitpath(14);
```

טבלה 41–9: פלט של הפונקציה fn_splitpath

pos	*empid*
1	1
2	3
3	7
4	9
5	14

כעת כדי לקבל את שרשרת הניהול של עובד נתון, בצע join בין הטבלה המוחזרת על ידי הפונקציה לבין טבלת Employees:

```
SELECT REPLICATE(' | ', lvl) + empname
FROM dbo.fn_splitpath(14) AS SP
  JOIN dbo.Employees AS E
    ON E.empid = SP.empid
ORDER BY path;
```

סטים מקוננים (Nested Sets)

סטים מקוננים הם אחד מהפתרונות הכי אלגנטיים שראיתי אי פעם לעיצוב עצים.

מידע נוסף: ג'ו סלקו (Joe Celko) מציג דיון מקיף של המודל Nested Sets בכתביו. תוכל למצוא את הדיון של ג'ו סלקו בנושא סטים מקוננים בספר שלו, Joe Celko's Trees and Hierarchies in SQL for Smarties (Morgan-Kaufmann, 2004).

כאן אציג יישומי T-SQL של המודל, אשר ברובם עובדים ב– SQL Server 2005 בלבד, מכיוון שהם משתמשים במאפיינים חדשים כמו CTEs רקורסיביים והפונקציה ROW_NUMBER.

היתרונות העיקריים של פתרון הסטים המקוננים הם שאילתות פשוטות ומהירות, אותן אתאר בהמשך, וכמו כן העובדה שבפתרון זה אין הגבלת רמות. עם זאת, עם סטים גדולים של נתונים, מעשיות הפתרון לרוב מוגבלת לעצים סטטיים. עבור סביבות דינמיות, הפתרון מוגבל לעצים קטנים (ואפשרי גם ליערות גדולים, אך כאלו שמורכבים מעצים קטנים).

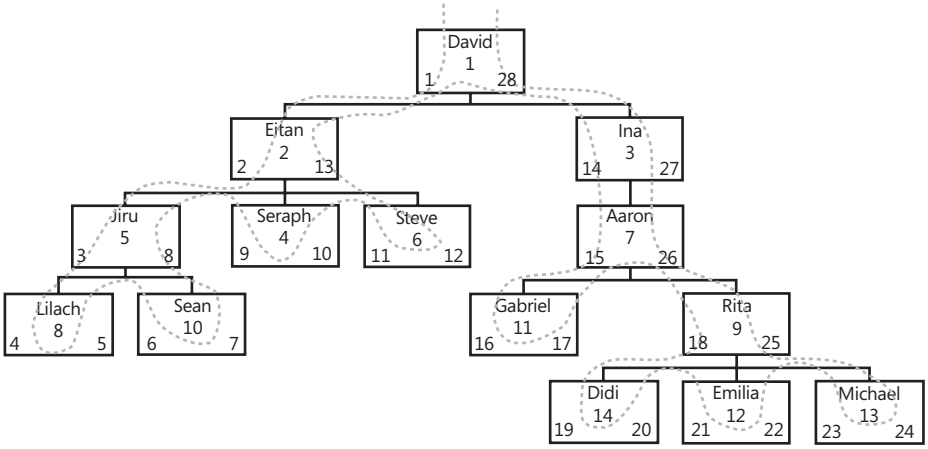
במקום לייצג עץ כרשימת סמיכויות (יחסי הורה/בן), פתרון זה מעצב את היחסים בעץ כסטים מקוננים. הורה מיוצג במודל הסטים המקוננים כסט מכיל ובן כסט מוכל. יחסי הכלה בסט מיוצגים עם שני ערכים שלמים המוקצים לכל סט: שמאל וימין. לכל הסטים: הערך השמאלי של הסט, קטן מכל הערכים השמאליים של הסטים המוכלים, והערך הימני של הסט גבוה

מכל הערכים הימניים של הסטים המוכלים. באופן טבעי, יחסי הכלה אלו הם טרנזיטיביים (transitive) במונחים של יחסי n רמות (אב/צאצא). השאילתות מבוססות על יחסי סטים מקוננים אלו. לוגית, זה כאילו סט פורש שתי זרועות סביב כל הסטים המוכלים שלו.

הקצאת ערכי שמאל וימין

תרשים 5–9 מספק ייצוג גרפי של היררכיית Employees עם ערכי שמאל וימין מוקצים לכל עובד.

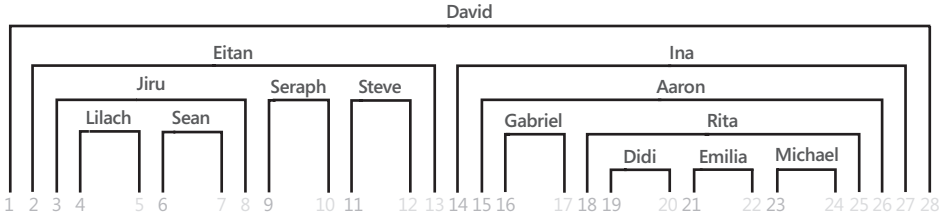
תרשים 5–9: היררכיית Employees כסטים מקוננים



הקו המתפתל העובר לאורך העץ מייצג את סדר ההקצאה של ערכי השמאל והימין. שים לב שהמודל מאפשר לך לבחור באיזה סדר להקצות ערכים לאחים. במקרה המסוים הזה, בחרתי לסרוק אחים לפי סדר שם עובד.

אתה מתחיל בשורש, וסורק את העץ נגד כיוון השעון. בכל פעם שאתה נכנס לצומת, אתה מגדיל מונה וקובע אותו כערך השמאלי של הצומת. בכל פעם שאתה עוזב צומת, אתה מגדיל את המונה וקובע אותו כערך הימני של הצומת. אלגוריתם זה יכול להיות מיושם עד לפרט האחרון כרוטינה איטרטיבית/רקורסיבית מקצה לכל צומת ערכי שמאל וימין. עם זאת, יישום כזה דורש סריקה של העץ צומת בכל פעם, דבר שעשוי להיות איטי מאוד. אני אציג אלגוריתם מהיר יותר הסורק את העץ רמה בכל פעם. האלגוריתם המרכזי מבוסס על לוגיקה בה דנתי מוקדם יותר בפרק, סריקת העץ רמה בכל פעם וחישוב מסלולי מיון בינאריים. כדי להבין אלגוריתם זה, יעזור לך אם תבחן את תרשים 6–9.

תרשים 6–9: איור מודל סטים מקוננים



התרשים מראה כל עובד כפורש שתי זרועות סביב הכפופים לו. ערכי שמאל וימין יכולים כעת להיות מוקצים לזרועות השונות, על ידי הגדלת מונה משמאל לימין. זכור איור זה, שכן הוא המפתח להבנת הפתרון שאציג.

שוב, נקודת הבסיס הוא האלגוריתם המקורי הסורק תת–עץ, רמה בכל פעם ובונה מסלול מיון בינארי בהתבסס על מיון אחים רצוי (לדוגמה, empname, empid).

שים לב: כדי לקבל ביצועים טובים, עליך ליצור אינדקס על קוד ההורה ועל טורי המיון של העובד – לדוגמה, mgrid, empname, empid.

במקום לייצר שורה אחת לכל צומת (כמו שהיה המקרה בפתרונות הקודמים – ליצור ערכי מיון בהתבסס על מסלול בינארי), אתה מייצר שתי שורות על ידי ביצוע cross-join בין כל רמה לבין טבלת עזר בעלת שני מספרים: n=1 המייצג את הזרוע השמאלית, ו–n=2 המייצג את הזרוע הימנית. עדיין, המסלולים הבינאריים נבנים ממספרי שורה, אך במקרה זה מספר הזרוע נלקח בחשבון מלבד האלמנטים האחרים של המיון (לדוגמה empname, empid, n). השאילתה המחזירה את רמת הכפופים הבאה, מחזירה את הכפופים של הזרוע השמאלית בלבד – שוב, מתבצע cross-join עם שני מספרים (n=1, n=2) כדי לייצר שתי זרועות לכל צומת.

בקטע–קוד 30–9, יש יישום CTE של אלגוריתם זה והוא מפיק את הפלט המוצג בטבלה 42–9. המטרה של קוד זה היא לייצר שני מסלולי מיון בינאריים לכל עובד, שמאוחר יותר ישמשו לחישוב ערכי שמאל וימין. בטרם תריץ קוד זה, ודא שיש לך את טבלת Employees המקורית במסד הנתונים tempdb. אם אין לך, ראשית הרץ שוב את הקוד בקטע–קוד 1–9.

קטע–קוד 30–9: ייצור מסלולי מיון בינאריים המייצגים יחסי סטים מקוננים

```
USE tempdb;
GO
-- Create index to speed sorting siblings by empname, empid
CREATE UNIQUE INDEX idx_unc_mgrid_empname_empid
  ON dbo.Employees(mgrid, empname, empid);
GO
```

```
DECLARE @root AS INT;
SET @root = 1;

-- CTE with two numbers: 1 and 2
WITH TwoNumsCTE
AS
(
  SELECT 1 AS n UNION ALL SELECT 2
),
-- CTE with two binary sort paths for each node:
--   One smaller than descendants sort paths
--   One greater than descendants sort paths
SortPathCTE
AS
(
  SELECT empid, 0 AS lvl, n,
    CAST(n AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees CROSS JOIN TwoNumsCTE
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, P.lvl + 1, TN.n,
    P.sortpath + CAST(
      (-1+ROW_NUMBER() OVER(PARTITION BY C.mgrid
                          -- *** determines order of siblings ***
                          ORDER BY C.empname, C.empid))/2*2+TN.n
      AS BINARY(4))
  FROM SortPathCTE AS P

    JOIN dbo.Employees AS C
      ON P.n = 1
      AND C.mgrid = P.empid
    CROSS JOIN TwoNumsCTE AS TN
)
SELECT * FROM SortPathCTE
ORDER BY sortpath;
```

טבלה 42–9: מסלולי מיון בינאריים המייצגים יחסי סטים מקוננים

empid	*lvl*	*n*	*sortpath*
1	0	1	0x00000001
2	1	1	0x0000000100000001
5	2	1	0x000000010000000100000001
8	3	1	0x00000001000000010000000100000001
8	3	2	0x00000001000000010000000100000002
10	3	1	0x00000001000000010000000100000003
10	3	2	0x00000001000000010000000100000004
5	2	2	0x000000010000000100000002
4	2	1	0x000000010000000100000003
4	2	2	0x000000010000000100000004
6	2	1	0x000000010000000100000005
6	2	2	0x000000010000000100000006
2	1	2	0x0000000100000002
3	1	1	0x0000000100000003
7	2	1	0x000000010000000300000001
11	3	1	0x00000001000000030000000100000001
11	3	2	0x00000001000000030000000100000002
9	3	1	0x00000001000000030000000100000003
14	4	1	0x0000000100000003000000010000000300000001
14	4	2	0x0000000100000003000000010000000300000002
12	4	1	0x0000000100000003000000010000000300000003
12	4	2	0x0000000100000003000000010000000300000004
13	4	1	0x0000000100000003000000010000000300000005
13	4	2	0x0000000100000003000000010000000300000006
9	3	2	0x00000001000000030000000100000004
7	2	2	0x000000010000000300000002
3	1	2	0x0000000100000004
1	0	2	0x00000002

TwoNumsCTE היא טבלת עזר עם שני מספרים המייצגים את שתי הזרועות. כמובן, תוכל להשתמש בטבלת Nums אמיתית אם תרצה, במקום לייצר טבלה וירטואלית.

שני מסלולי מיון נוצרים לכל צומת. השמאלי מיוצג על ידי n=1, והימני על ידי n=2. שים לב שעבור צומת נתון, מסלול המיון השמאלי קטן מכל מסלולי המיון השמאליים של הכפופים, ומסלול המיון הימני גדול מכל מסלולי המיון הימניים של הכפופים. מסלולי המיון ישמשו ליצירת ערכי השמאל והימין בתרשים 6–9. עליך לייצר ערכי שמאל וימין שלמים, כדי לייצג את יחסי הסטים המקוננים בין העובדים. כדי להקצות את הערכים השלמים לזרועות (sortval), השתמש בפונקציה ROW_NUMBER בהתבסס על סדר sortpath. לבסוף, כדי להחזיר שורה אחת לכל עובד, המכילה את ערכי שמאל וימין השלמים, קבץ את השורות לפי עובד ורמה, והחזר את MIN(sortval) כערך השמאלי ואת MAX(sortval) כערך הימני. הפתרון המלא ליצירת ערכי שמאל וימין מוצג בקטע–קוד 9–31 ומפיק את הפלט המוצג בטבלה 9–43.

קטע–קוד 9–31: קוד CTE היוצר יחסי סטים מקוננים

```
DECLARE @root AS INT;
SET @root = 1;

-- CTE with two numbers: 1 and 2
WITH TwoNumsCTE
AS
(
  SELECT 1 AS n UNION ALL SELECT 2
),
-- CTE with two binary sort paths for each node:
--   One smaller than descendants sort paths
--   One greater than descendants sort paths
SortPathCTE
AS
(
  SELECT empid, 0 AS lvl, n,
    CAST(n AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees CROSS JOIN TwoNumsCTE
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, P.lvl + 1, TN.n,
    P.sortpath + CAST(
      (-1+ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        -- *** determines order of siblings ***
                        ORDER BY C.empname, C.empid))/2*2+TN.n
      AS BINARY(4))
```

```
  FROM SortPathCTE AS P
    JOIN dbo.Employees AS C
      ON P.n = 1
      AND C.mgrid = P.empid
    CROSS JOIN TwoNumsCTE AS TN
),
-- CTE with Row Numbers Representing sortpath Order
SortCTE
AS
(
  SELECT empid, lvl,
    ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval
  FROM SortPathCTE
),
-- CTE with Left and Right Values Representing
-- Nested Sets Relationships
NestedSetsCTE
AS
(
  SELECT empid, lvl, MIN(sortval) AS lft, MAX(sortval) AS rgt
  FROM SortCTE
  GROUP BY empid, lvl
)
SELECT * FROM NestedSetsCTE
ORDER BY lft;
```

יישום אלגוריתם זה ב– SQL Server 2000 דומה, אך הוא ארוך ואיטי יותר, בעיקר בגלל החישוב של מספרי שורה על ידי שימוש בערכי identity במקום על ידי הפונקציה ROW_NUMBER. עליך לממש תוצאות ביניים בטבלה כדי לייצר ערכי identity. לשם הפשטות, אציג פתרון עם UDF, כאשר אחים ממוינים לפי empname, empid. כדי ליצור את ה–UDF fn_empsnestedsets, הרץ את הקוד בקטע–קוד 32–9.

טבלה 9–43: ערכי שמאל וימין מיוצרים עם CTE

empid	*lvl*	*lft*	*rgt*
1	0	1	28
2	1	2	13
5	2	3	8
8	3	4	5
10	3	6	7
4	2	9	10
6	2	11	12
3	1	14	27
7	2	15	26
11	3	16	17
9	3	18	25
14	4	19	20
12	4	21	22
13	4	23	24

קטע–קוד 9–32: קוד יצירה של הפונקציה fn_empsnestedsets

```
-----------------------------------------------------------------------
-- Function: fn_empsnestedsets, Nested Sets Relationships
--
-- Input   : @root INT: Root of subtree
--
-- Output  : @NestedSets Table: employee id, level in the subtree,
--                              left and right values representing
--                              nested sets relationships
--
-- Process : * Loads subtree into @SortPath,
--             first root, then a level at a time.
--             Note: two instances of each employee are loaded;
--                   one representing left arm (n = 1),
--                   and one representing right (n = 2).
--             For each employee and arm, a binary path is constructed,
--             representing the nested sets position.
--             The binary path has 4 bytes for each of the employee's
--             ancestors. For each ancestor, the 4 bytes represent
--             its position in the level (calculated with identity).
```

```
--             Finally @SortPath will contain a pair of rows for each
--             employee along with a sort path representing the arm's
--             nested sets position.
--           * Next, the rows from @SortPath are loaded
--             into @SortVals, sorted by sortpath. After the load,
--             an integer identity column sortval holds sort values
--             representing the nested sets position of each arm.
--           * The data from @SortVals is grouped by employee,
--             generating the left and right values for each employee
--             in one row. The result set is loaded into the
--             @NestedSets table, which is the function's output.
--
---------------------------------------------------------------------
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.fn_empsnestedsets') IS NOT NULL
  DROP FUNCTION dbo.fn_empsnestedsets;
GO
CREATE FUNCTION dbo.fn_empsnestedsets(@root AS INT)
  RETURNS @NestedSets TABLE
(
  empid INT NOT NULL PRIMARY KEY,
  lvl   INT NOT NULL,
  lft   INT NOT NULL,
  rgt   INT NOT NULL
)
AS
BEGIN
  DECLARE @lvl AS INT;
  SET @lvl = 0;

  -- @TwoNums: Table Variable with two numbers: 1 and 2
  DECLARE @TwoNums TABLE(n INT NOT NULL PRIMARY KEY);
  INSERT INTO @TwoNums(n) SELECT 1 AS n UNION ALL SELECT 2;
  -- @SortPath: Table Variable with two binary sort paths
  -- for each node:
  --   One smaller than descendants sort paths
  --   One greater than descendants sort paths
  DECLARE @SortPath TABLE
  (
    empid    INT            NOT NULL,
```

```
  lvl      INT            NOT NULL,
  n        INT            NOT NULL,
  sortpath VARBINARY(900) NOT NULL,
  rownum   INT            NOT NULL IDENTITY,
  UNIQUE(lvl, n, empid)
);

-- Load root into @SortPath
INSERT INTO @SortPath(empid, lvl, n, sortpath)
  SELECT empid, @lvl, n,
    CAST(n AS BINARY(4)) AS sortpath
  FROM dbo.Employees CROSS JOIN @TwoNums
  WHERE empid = @root

WHILE @@rowcount > 0
BEGIN
  SET @lvl = @lvl + 1;

  -- Load next level into @SortPath
  INSERT INTO @SortPath(empid, lvl, n, sortpath)
    SELECT C.empid, @lvl, TN.n, P.sortpath
    FROM @SortPath AS P
      JOIN dbo.Employees AS C
        ON  P.lvl = @lvl - 1
        AND P.n = 1
        AND C.mgrid = P.empid
      CROSS JOIN @TwoNums AS TN
    -- *** Determines order of siblings ***
    ORDER BY C.empname, C.empid, TN.n;

  -- Update sort path to include child's position
  UPDATE @SortPath
    SET sortpath = sortpath + CAST(rownum AS BINARY(4))
  WHERE lvl = @lvl;
END

-- @SortVals: Table Variable with row numbers
-- representing sortpath order
DECLARE @SortVals TABLE
(
  empid     INT NOT NULL,
  lvl       INT NOT NULL,
  sortval   INT NOT NULL IDENTITY
)
```

```
  -- Load data from @SortPath sorted by sortpath
  -- to generate sort values
  INSERT INTO @SortVals(empid, lvl)
    SELECT empid, lvl FROM @SortPath ORDER BY sortpath;

  -- Load data into @NestedSets, generating left and right
  -- values representing nested sets relationships
  INSERT INTO @NestedSets(empid, lvl, lft, rgt)
    SELECT empid, lvl, MIN(sortval), MAX(sortval)
    FROM @SortVals
    GROUP BY empid, lvl

  RETURN;
END
GO
```

כדי לבדוק את הפונקציה, הרץ את הקוד הבא, המייצר את הפלט המוצג בטבלה 9–44:

```
SELECT * FROM dbo.fn_empsnestedsets(1)
ORDER BY lft;
```

טבלה 9–44: ערכי שמאל וימין מיוצרים על ידי UDF

empid	*lvl*	*lft*	*rgt*
1	0	1	28
2	1	2	13
5	2	3	8
8	3	4	5
10	3	6	7
4	2	9	10
6	2	11	12
3	1	14	27
7	2	15	26
11	3	16	17
9	3	18	25
14	4	19	20
12	4	21	22
13	4	23	24

בפסקת הפתיחה של הסעיף "סטים מקוננים", ציינתי שפתרון זה אינו מתאים לעצים דינמיים גדולים (עצים בהם מתקיימים שינויים תכופים). נניח שאחסנת ערכי שמאל וימין בשני טורים נוספים בטבלת Employees. שים לב שלא תצטרך יותר את הטור mgrid בטבלה, שכן שני הטורים הנוספים עם ערכי השמאל והימין מספיקים כדי לענות על בקשות לכפופים, אבות, וכך הלאה. חשוב על אפקט הזעזוע של הוספת צומת לעץ. לדוגמה, התבונן בתרשימים 5–9 ו–6–9, ונסה לחשוב על ההשפעה של הוספת כפוף חדש לסטיב (Steve). לסטיב יש ערכי שמאל וימין 11 ו–12, בהתאמה. הערך הימני של סטיב, ולמעשה כל ערכי שמאל וימין בעץ שהיו גדולים או שווים ל–14, צריכים לגדול בשתיים. בממוצע, לפחות מחצית מהצמתים בעץ חייבים להיות מעודכנים בכל פעם שנוסף צומת חדש. כפי שתוכל לראות, אפקט הזעזוע כאן משמעותי ביותר. זוהי הסיבה שפתרון הסטים המקוננים מתאים לעץ גדול רק אם הוא סטטי, או אם עליך להריץ מדי פעם שאילתות מול snapshot סטטי של העץ.

סטים מקוננים יכולים לספק ביצועים סבירים עם עצים דינמיים קטנים (או יערות עם עצים קטנים) – לדוגמה, תחזוקת דיונים בפורומים כאשר כל דיון הוא עץ קטן עצמאי בתוך יער. תוכל ליישם פתרון המסנכרן את ערכי השמאל והימין של העץ עם כל שינוי. תוכל לבצע זאת על ידי שימוש בפרוצדורה מאוחסנת, או אפילו בטריגרים, כל עוד עלות השינוי קטנה מספיק כדי להיות נסבלת. לא אכנס אפילו לווריאציות על מודל הסטים המקוננים המתחזקים פערים בין ערכים (כלומר, משאירים מקום להוספת עלים חדשים ללא עבודה כה רבה), שכן כולם בסופו של דבר מוגבלים.

כדי לייצר טבלת עובדים (EmployeesNS) עם קוד העובד, שם העובד, משכורת, רמה, ערכי ימין ושמאל, בצע join על השאילתה החיצונית, או של ה–CTE או של ה–UDF, והשתמש במשפט INSERT SELECT. הרץ את הקוד בקטע–קוד 33–9 כדי ליצור את טבלת EmployeesNS עם אחים ממוינים לפי empname, empid.

קטע–קוד 33–9: מימוש יחסי סטים מקוננים בטבלה

```
SET NOCOUNT ON;
USE tempdb;
GO

DECLARE @root AS INT;
SET @root = 1;

WITH TwoNumsCTE
AS
(
  SELECT 1 AS n UNION ALL SELECT 2
),
```

```
SortPathCTE
AS
(
  SELECT empid, 0 AS lvl, n,
    CAST(n AS VARBINARY(MAX)) AS sortpath
  FROM dbo.Employees CROSS JOIN TwoNumsCTE
  WHERE empid = @root

  UNION ALL

  SELECT C.empid, P.lvl + 1, TN.n,
    P.sortpath + CAST(
      ROW_NUMBER() OVER(PARTITION BY C.mgrid
                        -- *** determines order of siblings ***
                        ORDER BY C.empname, C.empid, TN.n)
      AS BINARY(4))
  FROM SortPathCTE AS P
    JOIN dbo.Employees AS C
      ON P.n = 1
      AND C.mgrid = P.empid
    CROSS JOIN TwoNumsCTE AS TN
),
SortCTE
AS
(
  SELECT empid, lvl,
    ROW_NUMBER() OVER(ORDER BY sortpath) AS sortval
  FROM SortPathCTE
),
NestedSetsCTE
AS
(
  SELECT empid, lvl, MIN(sortval) AS lft, MAX(sortval) AS rgt
  FROM SortCTE
  GROUP BY empid, lvl
)
SELECT E.empid, E.empname, E.salary, NS.lvl, NS.lft, NS.rgt
INTO dbo.EmployeesNS
FROM NestedSetsCTE AS NS
  JOIN dbo.Employees AS E
    ON E.empid = NS.empid;

ALTER TABLE dbo.EmployeesNS ADD PRIMARY KEY NONCLUSTERED(empid);
CREATE UNIQUE CLUSTERED INDEX idx_unc_lft_rgt ON dbo.EmployeesNS(lft, rgt);
GO
```

ביצוע שאילתות

טבלת EmployeesNS מעצבת עץ עובדים כסטים מקוננים. ביצוע שאילתות הוא פשוט, אלגנטי ומהיר עם האינדקס על ערכי השמאל והימין.

בסעיף הבא אציג בקשות נפוצות מול עץ ופתרון השאילתה לכל אחת, ולאחריו פלט השאילתה.

החזר את תת–העץ של שורש נתון, והפק את הפלט המוצג בטבלה 45–9:

```
SELECT C.empid, REPLICATE(' | ', C.lvl - P.lvl) + C.empname AS empname
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON P.empid = 3
    AND C.lft >= P.lft AND C.rgt <= P.rgt
ORDER BY C.lft;
```

טבלה 45–9: תת–עץ של שורש נתון

empid	*empname*
3	Ina
7	| Aaron
11	| | Gabriel
9	| | Rita
14	| | | Didi
12	| | | Emilia
13	| | | Michael

השאילתה מבצעת join בין שני מופעים של EmployeesNS. אחד מייצג את ההורה (P) ומסונן על ידי השורש הנתון. השני מייצג את הבן (C). שני המופעים מאוחדים בהתבסס על כך שערך השמאל של הבן גדול או שווה לערך השמאל של ההורה, וערך הימין של הבן, קטן או שווה לערך הימין של ההורה. אינדנטציה של הפלט מושגת על ידי שכפול מחרוזת (' | ') מספר פעמים כרמת הבן פחות רמת ההורה. הפלט ממוין לפי ערך השמאל של הבן, אשר בהגדרה מייצג מיון היררכי נכון, ומיון רצוי של אחים. שאילתת תת–העץ משמשת כנקודת הפתיחה למרבית השאילתות הבאות.

אם אינך מעוניין לקבל את צומת השורש של תת–העץ בפלט, השתמש באופרטורים גדול (<) וקטן (>) במקום האופרטורים גדול–שווה (=<) וקטן–שווה (=>). עבור שאילתת תת–העץ, הוסף מסנן בתנאי ה–join המחזיר רק צמתים, בהם ההפרש בין רמת הבן לרמת ההורה קטן או שווה למספר הרמות המבוקש תחת השורש.

החזר את תת–העץ של שורש נתון, תוך הגבלה של 2 רמות כפופים תחת השורש, וייצר את הפלט המוצג בטבלה 9–46:

```
SELECT C.empid, REPLICATE(' | ', C.lvl - P.lvl) + C.empname AS empname
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON P.empid = 3
    AND C.lft >= P.lft AND C.rgt <= P.rgt
    AND C.lvl - P.lvl <= 2
ORDER BY C.lft;
```

טבלה 9–46: תת–עץ של שורש נתון, עם הגבלת רמות

empid	*empname*
3	Ina
7	| Aaron
11	| | Gabriel
9	| | Rita

החזר צמתי עלה תחת שורש נתון, וייצר את הפלט המוצג בטבלה 9–47:

```
SELECT C.empid, C.empname
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON P.empid = 3
    AND C.lft >= P.lft AND C.rgt <= P.rgt
    AND C.rgt - C.lft = 1;
```

טבלה 9–47: צמתי עלים תחת שורש נתון

empid	*empname*
11	Gabriel
12	Emilia
13	Michael
14	Didi

צומת עלה הוא צומת עבורו הערך הימני גדול מהערך השמאלי ב–1 (אין כפופים). הוסף מסנן זה לתנאי ה–join של שאילתת תת–העץ. כפי שתוכל לראות, פתרון הסטים המקוננים מאפשר זיהוי מהיר יותר משמעותית של צמתי עלה מאשר פתרונות אחרים המשתמשים בביטוי NOT EXISTS.

החזר את מספר הכפופים של כל צומת, וייצר את הפלט המוצג בטבלה 9–48:

```
SELECT empid, (rgt - lft - 1) / 2 AS cnt,
  REPLICATE(' | ', lvl) + empname AS empname
FROM dbo.EmployeesNS
ORDER BY lft;
```

טבלה 9–48: מספר כפופים של כל צומת

empid	*cnt*	*empname*
1	13	David
2	5	| Eitan
5	2	| | Jiru
8	0	| | | Lilach
10	0	| | | Sean
4	0	| | Seraph
6	0	| | Steve
3	6	| Ina
7	5	| | Aaron
11	0	| | | Gabriel
9	3	| | | Rita
14	0	| | | | Didi
12	0	| | | | Emilia
13	0	| | | | Michael

מכיוון שלכל צומת בדיוק שני ערכי lft ו–rgt, וביישום שלנו לא קיימים פערים, באפשרותך לחשב את מספר הכפופים על ידי גישה אך ורק לצומת השורש של תת–העץ. המספר הוא: (rgt – lft – 1) / 2.

החזר את כל האבות של צומת נתון, וייצר את הפלט המוצג בטבלה 9–49:

```
SELECT P.empid, P.empname, P.lvl
FROM dbo.EmployeesNS AS P
  JOIN dbo.EmployeesNS AS C
    ON C.empid = 14
    AND C.lft >= P.lft AND C.rgt <= P.rgt;
```

טבלה 49–9: אבות של צומת נתון

empid	*empname*	*lvl*
1	David	0
3	Ina	1
7	Aaron	2
9	Rita	3
14	Didi	4

שאילתת האבות זהה כמעט לחלוטין לשאילתת תת–העץ. יחסי הסטים המקוננים נשארים זהים. ההבדל היחיד הוא שכאן אתה מסנן קוד צומת בן ספציפי, בעוד שבשאילתת תת–העץ סיננת קוד צומת הורה ספציפי.

כשתסיים לבצע שאילתות על טבלת EmployeesNS, אל תשכח להסיר אותה:

```
DROP TABLE dbo.EmployeesNS;
```

Transitive Closure

ה– transitive closure של גרף מכוון G (directed graph) הוא גרף המכיל מקצוע עבור כל זוג צמתים מ–G שיש ביניהם מסלול. כלומר ה– transitive closure מכיל את אותם צמתים כמו ב–G (זוגות צמתים שיש ביניהם קשר ישיר), ובנוסף את כל זוגות הצמתים שיש ביניהם קשר עקיף (מסלול). ה– transitive closure עוזר לענות על מספר שאלות מיידית, ללא הצורך לחקור מסלולים בגרף; לדוגמה, האם דוד הוא המנהל של אהרון (ישיר או עקיף)? אם ה– transitive closure של גרף Employees מכיל מקצוע מדוד לאהרון – הוא מנהל שלו. האם אספרסו כפול מכיל מים? האם אני יכול לנהוג מלוס–אנג'לס לניו–יורק? אם גרף הקלט מכיל את המקצועות (a, b) ו–(b, c), קיימים יחסים טרנזיטיביים (transitive relationship) בין a ל–c. ה– transitive closure יכיל את המקצועות (a, b), (b, c), וגם (a, c). אם דוד הוא המנהל הישיר של אינה, ואינה היא המנהלת הישירה של אהרון, דוד הוא טרנזיטיבית המנהל של אהרון, או שאהרון כפוף טרנזיטיבית לדוד.

קיימות בעיות המתייחסות ל– transitive closure, העוסקות במקרים מיוחדים של יחסים טרנזיטיביים. דוגמה לכך היא בעיית "המסלול הקצר ביותר" (הידועה גם כ"בעיית הסוכן הנוסע"), בה אתה מנסה לקבוע מהו המסלול הקצר ביותר בין שני צמתים. לדוגמה, מהו המסלול הקצר ביותר בין לוס–אנג'לס לניו–יורק?

בסעיף זה, אתאר פתרונות איטרטיביים/רקורסיביים לבעיות transitive closure והמסלול הקצר ביותר. בכמה מהדוגמאות שאביא, אשתמש ב–CTEs המתאימים ל– SQL Server 2005. כמו בדוגמאות שהצגתי קודם לכן בפרק, תוכל לבצע התאמות וליישם אלגוריתמים דומים ב– SQL Server 2000 על ידי שימוש ב–UDFs או בפרוצדורות מאוחסנות.

שים לב: הביצועים של כמה מהפתרונות שאציג (במיוחד אלו המשתמשים ב–CTEs רקורסיביים) מתדרדרים אקספוננציאלית ככל שגרף הקלט גדל. אציג אותם למטרות הדגמה מכיוון שהם פשוטים וטבעיים למדי. הם מתאימים לגרפים קטנים יחסית. קיימים אלגוריתמים יעילים לבעיות transitive closure (לדוגמה, האלגוריתמים של Floyd ו–Warshall) היכולים להיות מיושמים כאיטרציות של "רמה בכל פעם" (לרוחב תחילה). לפרטים על אלו, אנא פנה לכתובת http://www.nist.gov/dads. אציג פתרונות יעילים שסופקו על ידי סטיב קאס אשר יכולים להיות מיושמים בגרפים גדולים יותר.

גרף מכוון לא–מעגלי (Directed Acyclic Graph)

הבעיה הראשונה בה אדון היא יצירת transitive closure של גרף מכוון לא–מעגלי (DAG). בהמשך אראה לך כיצד להתמודד גם עם גרפים לא–מכוונים ומעגליים. העובדה האם הגרף מכוון או בלתי–מכוון אינה מסבכת את הפתרון משמעותית, בעוד שהתמודדות עם גרפים מעגליים כן. DAG הקלט בו אשתמש בדוגמה שלי, הוא עץ המוצר בו השתמשתי מוקדם יותר בפרק, אותו אתה יוצר על ידי הרצת הקוד בקטע–קוד 2–9.

הקוד המייצר את ה– transitive closure של עץ מוצר, דומה במידת מה לפתרונות לבעיית תת–הגרף (כלומר, אתה משתמש בשיטות חיפוש לרוחב תחילה). עם זאת, במקום להחזיר רק צומת שורש כאן, איבר העוגן מחזיר את כל היחסים של הרמה–הראשונה בעץ המוצר. במרבית הגרפים, המשמעות היא כל זוגות מקור/יעד הקיימים. במקרה שלנו, המשמעות היא כל זוגות מוצר/פריט בהם המוצר אינו NULL. האיבר הרקורסיבי מבצע join בין ה–CTE המייצג את הרמה הקודמת או ההורה (P) לבין עץ המוצר המייצג את הרמה הבאה או הבן (C). הוא מחזיר את קוד המוצר המקורי (P) כמקור, ואת קוד מוצר הבן (C) כיעד. השאילתה החיצונית מחזירה את זוגות מוצר/פריט הייחודיים. זכור שמספר מסלולים עשויים להוביל לפריט בעץ המוצר, אך עליך להחזיר כל זוג מובחן פעם אחת בלבד.

הרץ את הקוד בקטע–קוד 9–34 כדי ליצור את ה– transitive closure של עץ המוצר המוצג בטבלה 9–50.

קטע–קוד 9–34: Transitive closure של עץ מוצר (DAG)

```
WITH BOMTC
AS
(
  -- Return all first-level containment relationships
  SELECT assemblyid, partid
  FROM dbo.BOM
  WHERE assemblyid IS NOT NULL

  UNION ALL
```

```
  -- Return next-level containment relationships
  SELECT P.assemblyid, C.partid
  FROM BOMTC AS P
    JOIN dbo.BOM AS C
      ON C.assemblyid = P.partid
)
-- Return distinct pairs that have
-- transitive containment relationships
SELECT DISTINCT assemblyid, partid
FROM BOMTC;
```

טבלה 50–9: Transitive closure של עץ מוצר (DAG)

assemblyid	*partid*
1	6
1	7
1	10
1	13
1	14
2	6
2	7
2	10
2	11
2	13
2	14
3	6
3	7
3	11
3	12
3	14
3	16
3	17
4	9
4	12
4	14
4	16
4	17
5	9

assemblyid	*partid*
5	12
5	14
5	16
5	17
10	13
10	14
12	14
12	16
12	17
16	17

פתרון זה מעלים מקצועות (edges) כפולים שנמצאו ב–BOMCTE על ידי הפעלת פסוקית DISTINCT בשאילתה החיצונית. פתרון יעיל יותר יהיה להימנע בכלל מקבלת כפילויות על ידי שימוש בביטוי NOT EXISTS בשאילתה שרצה בלולאה; ביטוי כזה יסנן מקצועות חדשים שנמצא שאינם מופיעים בסט המקצועות שנמצאו כבר. עם זאת, יישום כזה לא יוכל להשתמש ב–CTE מכיוון שלאיבר הרקורסיבי ב–CTE יש גישה רק ל"רמה הקודמת המיידית", בניגוד ל"כל הרמות הקודמות" שהושגו עד כה. במקום זאת, תוכל להשתמש ב–UDF הקוראת לשאילתה שרצה בלולאה ומכניסה כל רמת צמתים שהושגה לתוך משתנה טבלה. הרץ את הקוד בקטע–קוד 35–9 כדי ליצור את ה– fn_BOMTC UDF, המיישמת לוגיקה זו.

קטע–קוד 35–9: קוד יצירה של ה– fn_BOMTC UDF

```
IF OBJECT_ID('dbo.fn_BOMTC') IS NOT NULL
  DROP FUNCTION dbo.fn_BOMTC;
GO

CREATE FUNCTION fn_BOMTC() RETURNS @BOMTC TABLE
(
  assemblyid  INT NOT NULL,
  partid      INT NOT NULL,
  PRIMARY KEY (assemblyid, partid)
)
AS
BEGIN
  INSERT INTO @BOMTC(assemblyid, partid)
    SELECT assemblyid, partid
    FROM dbo.BOM
    WHERE assemblyid IS NOT NULL
```

```
  WHILE @@rowcount > 0
    INSERT INTO @BOMTC
    SELECT P.assemblyid, C.partid
    FROM @BOMTC AS P
      JOIN dbo.BOM AS C
        ON C.assemblyid = P.partid
    WHERE NOT EXISTS
      (SELECT * FROM @BOMTC AS P2
       WHERE P2.assemblyid = P.assemblyid
       AND P2.partid = C.partid);

  RETURN;
END
GO
```

הרץ את הקוד הבא כדי לבצע שאילתה על הפונקציה ותקבל את הפלט המוצג בטבלה 9–50:

```
SELECT assemblyid, partid FROM fn_BOMTC();
```

אם ברצונך להחזיר את כל המסלולים בעץ המוצר, בצירוף המרחק ברמות בין הפריטים, תשתמש באלגוריתם דומה עם מספר תוספות ושינויים. אתה מחשב את המרחק באותה דרך בה אתה מחשב את ערך הרמה בפתרונות תת–הגרף/תת–העץ. כלומר, העוגן מקצה מרחק קבוע של 1 עבור הרמה הראשונה, והאיבר הרקורסיבי פשוט מוסיף 1 בכל איטרציה. כמו כן, חישוב המסלול דומה לזה בו השתמשנו בפתרונות תת–הגרף/תת–העץ. העוגן מייצר מסלול המורכב מ– source_id + '.' + target_id + '.'. האיבר הרקורסיבי מייצר אותו כ– parent's path + target_id + '.'. לבסוף, השאילתה החיצונית פשוט מחזירה את כל המסלולים (ללא הפעלת DISTINCT במקרה זה).

הרץ את הקוד בקטע–קוד 9–36 כדי לייצר את כל המסלולים האפשריים בעץ המוצר והמרחקים שלהם.

קטע–קוד 9–36: כל המסלולים בעץ מוצר

```
WITH BOMPaths
AS
(
  SELECT assemblyid, partid,
    1 AS distance, -- distance in first level is 1
    -- path in first level is .assemblyid.partid.
    '.' + CAST(assemblyid AS VARCHAR(MAX)) +
    '.' + CAST(partidAS VARCHAR(MAX)) + '.' AS path
```

```
  FROM dbo.BOM
  WHERE assemblyid IS NOT NULL

  UNION ALL

  SELECT P.assemblyid, C.partid,
    -- distance in next level is parent's distance + 1
    P.distance + 1,
    -- path in next level is parent_path.child_partid.
    P.path + CAST(C.partid AS VARCHAR(MAX)) + '.'
  FROM BOMPaths AS P
    JOIN dbo.BOM AS C
      ON C.assemblyid = P.partid
)
-- Return all paths
SELECT * FROM BOMPaths;
```

טבלה 51–9: כל המסלולים בעץ מוצר

assemblyid	*partid*	*distance*	*path*
1	6	1	.1.6.
2	6	1	.2.6.
3	6	1	.3.6.
1	7	1	.1.7.
2	7	1	.2.7.
3	7	1	.3.7.
4	9	1	.4.9.
5	9	1	.5.9.
1	10	1	.1.10.
2	10	1	.2.10.
2	11	1	.2.11.
3	11	1	.3.11.
3	12	1	.3.12.
4	12	1	.4.12.
5	12	1	.5.12.

assemblyid	*partid*	*distance*	*path*
10	13	1	.10.13.
1	14	1	.1.14.
2	14	1	.2.14.
10	14	1	.10.14.
12	14	1	.12.14.
12	16	1	.12.16.
16	17	1	.16.17.
12	17	2	.12.16.17.
5	14	2	.5.12.14.
5	16	2	.5.12.16.
5	17	3	.5.12.16.17.
4	14	2	.4.12.14.
4	16	2	.4.12.16.
4	17	3	.4.12.16.17.
3	14	2	.3.12.14.
3	16	2	.3.12.16.
3	17	3	.3.12.16.17.
2	13	2	.2.10.13.
2	14	2	.2.10.14.
1	13	2	.1.10.13.
1	14	2	.1.10.14.

כדי לבודד רק את המסלולים הקצרים ביותר, הוסף CTE שני (BOMMinDist) המקבץ את כל המסלולים לפי מוצר ופריט, ומחזיר את המרחק המינימלי לכל קבוצה. ובשאילתה החיצונית, בצע join בין ה–CTE הראשון (BOMPaths) לבין BOMMinDist, בהתבסס על התאמת assembly, part ו–distance כדי להחזיר את המסלולים בפועל.

הרץ את הקוד בקטע–קוד 37–9 כדי לייצר את המסלולים הקצרים ביותר בעץ המוצר, כפי שמוצג בטבלה 52–9.

קטע–קוד 37–9: מסלולים קצרים ביותר בעץ מוצר

```
WITH BOMPaths -- All paths
AS
(
  SELECT assemblyid, partid,
    1 AS distance,
    '.' + CAST(assemblyid   AS VARCHAR(MAX)) +
    '.' + CAST(partidAS VARCHAR(MAX)) + '.' AS path
  FROM dbo.BOM
  WHERE assemblyid IS NOT NULL

  UNION ALL

  SELECT P.assemblyid, C.partid,
    P.distance + 1,
    P.path + CAST(C.partid AS VARCHAR(MAX)) + '.'
  FROM BOMPaths AS P
    JOIN dbo.BOM AS C
      ON C.assemblyid = P.partid
),
BOMMinDist AS -- Minimum distance for each pair
(
  SELECT assemblyid, partid, MIN(distance) AS mindist
  FROM BOMPaths
  GROUP BY assemblyid, partid
)
-- Shortest path for each pair
SELECT BP.*
FROM BOMMinDist AS BMD
  JOIN BOMPaths AS BP
    ON BMD.assemblyid = BP.assemblyid
    AND BMD.partid = BP.partid
    AND BMD.mindist = BP.distance;
```

טבלה 52–9: מסלולים קצרים ביותר בעץ מוצר

assemblyid	*partid*	*distance*	*path*
1	6	1	.1.6.
2	6	1	.2.6.
3	6	1	.3.6.
1	7	1	.1.7.
2	7	1	.2.7.
3	7	1	.3.7.
4	9	1	.4.9.
5	9	1	.5.9.
1	10	1	.1.10.
2	10	1	.2.10.
2	11	1	.2.11.
3	11	1	.3.11.
3	12	1	.3.12.
4	12	1	.4.12.
5	12	1	.5.12.
10	13	1	.10.13.
1	14	1	.1.14.
2	14	1	.2.14.
10	14	1	.10.14.
12	14	1	.12.14.
12	16	1	.12.16.
16	17	1	.16.17.
12	17	2	.12.16.17.
5	14	2	.5.12.14.
5	16	2	.5.12.16.
5	17	3	.5.12.16.17.
4	14	2	.4.12.14.
4	16	2	.4.12.16.
4	17	3	.4.12.16.17.
3	14	2	.3.12.14.
3	16	2	.3.12.16.
3	17	3	.3.12.16.17.
2	13	2	.2.10.13.
1	13	2	.1.10.13.

גרף בלתי–מכוון מעגלי (Undirected Cyclic Graph)

על אף ש– transitive closure מוגדר עבור גרף מכוון, ניתן להגדיר ולייצר אותו עבור גרפים בלתי–מכוונים, כאשר כל מקצוע מייצג יחסים דו–כיווניים. בדוגמאות שלי, אשתמש בגרף Roads, אותו אתה מייצר וממלא בנתונים על ידי הרצת הקוד בקטע–קוד 9–3. כדי לראות ייצוג חזותי של Roads, בחן את תרשים 9–4. כדי להחיל את פתרונות ה– transitive closure והמסלול הקצר ביותר על Roads, ראשית המר אותו לגרף–מכוון על ידי יצירת שני מקצועות מכוונים מכל מקצוע קיים:

```
SELECT city1 AS from_city, city2 AS to_city FROM dbo.Roads
UNION ALL
SELECT city2, city1 FROM dbo.Roads
```

לדוגמה, המקצוע (JFK, ATL) בגרף הבלתי–מכוון יופיע כמקצועות (JFK, ATL) ו–(ATL, JFK) בגרף המכוון. הראשון מייצג את הדרך מניו–יורק לאטלנטה, והשני מייצג את הדרך מאטלנטה לניו–יורק.

מכיוון ש–Roads הוא גרף מעגלי, עליך גם להשתמש בלוגיקת זיהוי–המחזוריות שתיארתי מוקדם יותר בפרק, כדי להימנע מלסרוק מסלולים מעגליים. כשאתה חמוש בשיטות לייצור גרף מכוון מגרף בלתי–מכוון ולזיהוי מחזוריות, יש בידך את כל הכלים להם אתה זקוק כדי לייצר את ה– transitive closure של Roads.

הרץ את הקוד בקטע–קוד 9–38 כדי לייצר את ה– transitive closure של Roads, המוצג בטבלה 9–53.

קטע–קוד 9–38: Transitive closure של Roads (גרף מעגלי בלתי–מכוון)

```
WITH Roads2 -- Two rows for each pair (from-->to, to-->from)
AS
(
  SELECT city1 AS from_city, city2 AS to_city FROM dbo.Roads
  UNION ALL
  SELECT city2, city1 FROM dbo.Roads
),
RoadPaths AS
(
  -- Return all first-level reachability pairs
  SELECT from_city, to_city,
    -- path is needed to identify cycles
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL
```

```
  -- Return next-level reachability pairs
  SELECT F.from_city, T.to_city,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
  FROM RoadPaths AS F
    JOIN Roads2 AS T
      -- if to_city appears in from_city's path, cycle detected
      ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
              THEN 1 ELSE 0 END = 0
      AND F.to_city = T.from_city
)
-- Return Transitive Closure of Roads
SELECT DISTINCT from_city, to_city
FROM RoadPaths;
```

טבלה 53–9: Transitive Closure של Roads

from to	*from to*	*from to*	*from to*	*from to*
ANC FAI	IAH DEN	LAX MCI	MIA ORD	SEA ATL
ATL DEN	IAH JFK	LAX MIA	MIA SEA	SEA DEN
ATL IAH	IAH LAX	LAX MSP	MIA SFO	SEA IAH
ATL JFK	IAH MCI	LAX ORD	MSP ATL	SEA JFK
ATL LAX	IAH MIA	LAX SEA	MSP DEN	SEA LAX
ATL MCI	IAH MSP	LAX SFO	MSP IAH	SEA MCI
ATL MIA	IAH ORD	MCI ATL	MSP JFK	SEA MIA
ATL MSP	IAH SEA	MCI DEN	MSP LAX	SEA MSP
ATL ORD	IAH SFO	MCI IAH	MSP MCI	SEA ORD
ATL SEA	JFK ATL	MCI JFK	MSP MIA	SEA SFO
ATL SFO	JFK DEN	MCI LAX	MSP ORD	SFO ATL
DEN ATL	JFK IAH	MCI MIA	MSP SEA	SFO DEN
DEN IAH	JFK LAX	MCI MSP	MSP SFO	SFO IAH
DEN JFK	JFK MCI	MCI ORD	ORD ATL	SFO JFK
DEN LAX	JFK MIA	MCI SEA	ORD DEN	SFO LAX
DEN MCI	JFK MSP	MCI SFO	ORD IAH	SFO MCI
DEN MIA	JFK ORD	MIA ATL	ORD JFK	SFO MIA
DEN MSP	JFK SEA	MIA DEN	ORD LAX	SFO MSP
DEN ORD	JFK SFO	MIA IAH	ORD MCI	SFO ORD
DEN SEA	LAX ATL	MIA JFK	ORD MIA	SFO SEA
DEN SFO	LAX DEN	MIA LAX	ORD MSP	
FAI ANC	LAX IAH	MIA MCI	ORD SEA	
IAH ATL	LAX JFK	MIA MSP	ORD SFO	

ה– Roads2 CTE יוצר את הגרף המכוון מתוך Roads. ה– RoadPaths CTE מחזיר את כל זוגות מקור/יעד האפשריים (דבר שגורם לקנס ביצועים גבוה), והוא נמנע מלהמשיך לסרוק מסלול עבורו זוהתה מחזוריות. השאילתה החיצונית מחזירה את כל זוגות מקור/יעד הייחודיים.

גם כאן באפשרותך להשתמש בלולאה במקום ב–CTE רקורסיבי כדי לבצע אופטימיזציה לפתרון, כפי שהדגמתי קודם בתרחיש עץ המוצר בקטע–קוד 35–9. הרץ את הקוד בקטע–קוד 39–9 כדי ליצור את ה– fn_RoadsTC UDF, המחזירה את ה– transitive closure של Roads על ידי שימוש בלולאות.

קטע–קוד 39–9: קוד יצירה של ה– fn_RoadsTC UDF

```
IF OBJECT_ID('dbo.fn_RoadsTC') IS NOT NULL
  DROP FUNCTION dbo.fn_RoadsTC;
GO

CREATE FUNCTION dbo.fn_RoadsTC() RETURNS @RoadsTC TABLE (
  from_city VARCHAR(3) NOT NULL,
  to_city   VARCHAR(3) NOT NULL,
  PRIMARY KEY (from_city, to_city)
)
AS
BEGIN
  DECLARE @added as INT;

  INSERT INTO @RoadsTC(from_city, to_city)
    SELECT city1, city2 FROM dbo.Roads;

  SET @added = @@rowcount;

  INSERT INTO @RoadsTC
    SELECT city2, city1 FROM dbo.Roads

  SET @added = @added + @@rowcount;

  WHILE @added > 0 BEGIN

    INSERT INTO @RoadsTC
      SELECT DISTINCT TC.from_city, R.city2
      FROM @RoadsTC AS TC
        JOIN dbo.Roads AS R
          ON R.city1 = TC.to_city
```

```
      WHERE NOT EXISTS
        (SELECT * FROM @RoadsTC AS TC2
         WHERE TC2.from_city = TC.from_city
           AND TC2.to_city = R.city2)
        AND TC.from_city <> R.city2;

    SET @added = @@rowcount;

    INSERT INTO @RoadsTC
      SELECT DISTINCT TC.from_city, R.city1
      FROM @RoadsTC AS TC
        JOIN dbo.Roads AS R
          ON R.city2 = TC.to_city
      WHERE NOT EXISTS
        (SELECT * FROM @RoadsTC AS TC2
         WHERE TC2.from_city = TC.from_city
           AND TC2.to_city = R.city1)
        AND TC.from_city <> R.city1;

    SET @added = @added + @@rowcount;
  END
  RETURN;
END
GO

-- Use the fn_RoadsTC UDF
SELECT * FROM dbo.fn_RoadsTC();
GO
```

הרץ את השאילתה הבאה כדי לקבל את ה– transitive closure של Roads המוצג בטבלה 9–53:

```
SELECT * FROM dbo.fn_RoadsTC();
```

כדי להחזיר את כל המסלולים והמרחקים, השתמש בלוגיקה דומה לזו בה השתמשנו בפתרון הגרף המכוון בסעיף הקודם. ההבדל כאן הוא שהמרחק אינו רק מונה רמות; אלא הוא הסכום של המרחקים לאורך המסלול מעיר אחת לאחרת.

הרץ את הקוד בקטע–קוד 9–40 כדי להחזיר את כל המסלולים והמרחקים ב–Roads.

קטע-קוד 9-40: כל המסלולים והמרחקים ב-Roads

```
WITH Roads2
AS
(
  SELECT city1 AS from_city, city2 AS to_city, distance FROM dbo.Roads
  UNION ALL
  SELECT city2, city1, distance FROM dbo.Roads
),
RoadPaths AS
(
  SELECT from_city, to_city, distance,
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  SELECT F.from_city, T.to_city, F.distance + T.distance,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
  FROM RoadPaths AS F
    JOIN Roads2 AS T
      ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
              THEN 1 ELSE 0 END = 0
      AND F.to_city = T.from_city
)
-- Return all paths and distances
SELECT * FROM RoadPaths;
```

לבסוף, כדי להחזיר את המסלול הקצר ביותר ב-Roads, השתמש באותה לוגיקה כמו בפתרון המסלול הקצר ביותר בגרף המכוון. הרץ את הקוד בקטע-קוד 9-41, כדי להחזיר את המסלולים הקצרים ביותר ב-Roads כפי שמוצג בטבלה 9-54.

קטע-קוד 9-41: מסלולים קצרים ביותר ב-Roads

```
WITH Roads2
AS
(
  SELECT city1 AS from_city, city2 AS to_city, distance FROM dbo.Roads
  UNION ALL
  SELECT city2, city1, distance FROM dbo.Roads
),
```

```
RoadPaths AS
(
  SELECT from_city, to_city, distance,
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  SELECT F.from_city, T.to_city, F.distance + T.distance,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
  FROM RoadPaths AS F
    JOIN Roads2 AS T
      ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
              THEN 1 ELSE 0 END = 0
      AND F.to_city = T.from_city
),
RoadsMinDist -- Min distance for each pair in TC
AS
(
  SELECT from_city, to_city, MIN(distance) AS mindist
  FROM RoadPaths
  GROUP BY from_city, to_city
)
-- Return shortest paths and distances
SELECT RP.*
FROM RoadsMinDist AS RMD
  JOIN RoadPaths AS RP
    ON RMD.from_city = RP.from_city
    AND RMD.to_city = RP.to_city
    AND RMD.mindist = RP.distance;
```

טבלה 54–9: מסלולים קצרים ביותר ב–Roads

from_city	*to_city*	*distance*	*path*
ANC	FAI	359	.ANC.FAI.
ATL	IAH	800	.ATL.IAH.
ATL	JFK	865	.ATL.JFK.
ATL	MCI	805	.ATL.MCI.
ATL	MIA	665	.ATL.MIA.
ATL	ORD	715	.ATL.ORD.
DEN	IAH	1120	.DEN.IAH.
DEN	LAX	1025	.DEN.LAX.
DEN	MCI	600	.DEN.MCI.
DEN	MSP	915	.DEN.MSP.
DEN	SEA	1335	.DEN.SEA.
DEN	SFO	1270	.DEN.SFO.
IAH	LAX	1550	.IAH.LAX.
IAH	MCI	795	.IAH.MCI.
IAH	MIA	1190	.IAH.MIA.
JFK	ORD	795	.JFK.ORD.
LAX	SFO	385	.LAX.SFO.
MCI	MSP	440	.MCI.MSP.
MCI	ORD	525	.MCI.ORD.
MSP	ORD	410	.MSP.ORD.
MSP	SEA	2015	.MSP.SEA.
SEA	SFO	815	.SEA.SFO.
FAI	ANC	359	.FAI.ANC.
IAH	ATL	800	.IAH.ATL.
JFK	ATL	865	.JFK.ATL.
MCI	ATL	805	.MCI.ATL.
MIA	ATL	665	.MIA.ATL.
ORD	ATL	715	.ORD.ATL.
IAH	DEN	1120	.IAH.DEN.

from_city	*to_city*	*distance*	*path*
LAX	DEN	1025	.LAX.DEN.
MCI	DEN	600	.MCI.DEN.
MSP	DEN	915	.MSP.DEN.
SEA	DEN	1335	.SEA.DEN.
SFO	DEN	1270	.SFO.DEN.
LAX	IAH	1550	.LAX.IAH.
MCI	IAH	795	.MCI.IAH.
MIA	IAH	1190	.MIA.IAH.
ORD	JFK	795	.ORD.JFK.
SFO	LAX	385	.SFO.LAX.
MSP	MCI	440	.MSP.MCI.
ORD	MCI	525	.ORD.MCI.
ORD	MSP	410	.ORD.MSP.
SEA	MSP	2015	.SEA.MSP.
SFO	SEA	815	.SFO.SEA.
SEA	ORD	2425	.SEA.MSP.ORD.
SEA	JFK	3220	.SEA.MSP.ORD.JFK.
ORD	SEA	2425	.ORD.MSP.SEA.
ORD	DEN	1125	.ORD.MCI.DEN.
ORD	IAH	1320	.ORD.MCI.IAH.
ORD	LAX	2150	.ORD.MCI.DEN.LAX.
ORD	SFO	2395	.ORD.MCI.DEN.SFO.
MSP	IAH	1235	.MSP.MCI.IAH.
SFO	IAH	1935	.SFO.LAX.IAH.
SFO	MIA	3125	.SFO.LAX.IAH.MIA.
MIA	LAX	2740	.MIA.IAH.LAX.
MIA	SFO	3125	.MIA.IAH.LAX.SFO.
LAX	MIA	2740	.LAX.IAH.MIA.
LAX	ATL	2350	.LAX.IAH.ATL.
SFO	MCI	1870	.SFO.DEN.MCI.
SFO	MSP	2185	.SFO.DEN.MSP.

from_city	*to_city*	*distance*	*path*
SFO	ORD	2395	.SFO.DEN.MCI.ORD.
SFO	ATL	2675	.SFO.DEN.MCI.ATL.
SFO	JFK	3190	.SFO.DEN.MCI.ORD.JFK.
SEA	IAH	2455	.SEA.DEN.IAH.
SEA	MCI	1935	.SEA.DEN.MCI.
SEA	ATL	2740	.SEA.DEN.MCI.ATL.
SEA	MIA	3405	.SEA.DEN.MCI.ATL.MIA.
MSP	LAX	1940	.MSP.DEN.LAX.
MSP	SFO	2185	.MSP.DEN.SFO.
MCI	LAX	1625	.MCI.DEN.LAX.
MCI	SEA	1935	.MCI.DEN.SEA.
MCI	SFO	1870	.MCI.DEN.SFO.
LAX	MCI	1625	.LAX.DEN.MCI.
LAX	MSP	1940	.LAX.DEN.MSP.
LAX	ORD	2150	.LAX.DEN.MCI.ORD.
LAX	JFK	2945	.LAX.DEN.MCI.ORD.JFK.
IAH	SEA	2455	.IAH.DEN.SEA.
ORD	MIA	1380	.ORD.ATL.MIA.
MIA	JFK	1530	.MIA.ATL.JFK.
MIA	MCI	1470	.MIA.ATL.MCI.
MIA	ORD	1380	.MIA.ATL.ORD.
MIA	MSP	1790	.MIA.ATL.ORD.MSP.
MIA	DEN	2070	.MIA.ATL.MCI.DEN.
MIA	SEA	3405	.MIA.ATL.MCI.DEN.SEA.
MCI	MIA	1470	.MCI.ATL.MIA.
JFK	IAH	1665	.JFK.ATL.IAH.
JFK	MIA	1530	.JFK.ATL.MIA.
IAH	JFK	1665	.IAH.ATL.JFK.
SEA	LAX	1200	.SEA.SFO.LAX.
MSP	ATL	1125	.MSP.ORD.ATL.
MSP	JFK	1205	.MSP.ORD.JFK.

from_city	*to_city*	*distance*	*path*
MSP	MIA	1790	.MSP.ORD.ATL.MIA.
MCI	JFK	1320	.MCI.ORD.JFK.
LAX	SEA	1200	.LAX.SFO.SEA.
JFK	MCI	1320	.JFK.ORD.MCI.
JFK	MSP	1205	.JFK.ORD.MSP.
JFK	SEA	3220	.JFK.ORD.MSP.SEA.
JFK	DEN	1920	.JFK.ORD.MCI.DEN.
JFK	LAX	2945	.JFK.ORD.MCI.DEN.LAX.
JFK	SFO	3190	.JFK.ORD.MCI.DEN.SFO.
IAH	MSP	1235	.IAH.MCI.MSP.
IAH	ORD	1320	.IAH.MCI.ORD.
IAH	SFO	1935	.IAH.LAX.SFO.
DEN	ORD	1125	.DEN.MCI.ORD.
DEN	ATL	1405	.DEN.MCI.ATL.
DEN	MIA	2070	.DEN.MCI.ATL.MIA.
DEN	JFK	1920	.DEN.MCI.ORD.JFK.
ATL	MSP	1125	.ATL.ORD.MSP.
ATL	DEN	1405	.ATL.MCI.DEN.
ATL	SEA	2740	.ATL.MCI.DEN.SEA.
ATL	SFO	2675	.ATL.MCI.DEN.SFO.
ATL	LAX	2350	.ATL.IAH.LAX.

כדי לספק מספר בקשות עבור המסלולים הקצרים ביותר בין שתי ערים, ייתכן שתרצה לממש את סט התוצאה בטבלה וליצור עליה אינדקס, כפי שמוצג בקטע–קוד 42–9:

קטע-קוד 42-9: טען מסלולי דרך קצרים ביותר לתוך טבלה

```
WITH Roads2
AS
(
  SELECT city1 AS from_city, city2 AS to_city, distance FROM dbo.Roads
  UNION ALL
  SELECT city2, city1, distance FROM dbo.Roads
),
RoadPaths AS
(
  SELECT from_city, to_city, distance,
    CAST('.' + from_city + '.' + to_city + '.' AS VARCHAR(MAX)) AS path
  FROM Roads2

  UNION ALL

  SELECT F.from_city, T.to_city, F.distance + T.distance,
    CAST(F.path + T.to_city + '.' AS VARCHAR(MAX))
  FROM RoadPaths AS F
    JOIN Roads2 AS T
      ON CASE WHEN F.path LIKE '%.' + T.to_city + '.%'
              THEN 1 ELSE 0 END = 0
      AND F.to_city = T.from_city
),
RoadsMinDist
AS
(
  SELECT from_city, to_city, MIN(distance) AS mindist
  FROM RoadPaths
  GROUP BY from_city, to_city
)
SELECT RP.*
INTO dbo.RoadPaths
FROM RoadsMinDist AS RMD
  JOIN RoadPaths AS RP
    ON RMD.from_city = RP.from_city
    AND RMD.to_city = RP.to_city
    AND RMD.mindist = RP.distance;
CREATE UNIQUE CLUSTERED INDEX idx_uc_from_city_to_city
  ON dbo.RoadPaths(from_city, to_city);
```

ברגע שהתוצאה ממומשת ונוצר עליה אינדקס, בקשה עבור המסלול הקצר ביותר בין שתי ערים יכולה להיענות מיידית. פתרון זה מעשי ומומלץ כאשר מידע משתנה לעיתים רחוקות. כמו שקורה לרוב, קיים שקלול בין "עדכני" ל"מהיר". השאילתה הבאה מבקשת את המסלול הקצר ביותר בין לוס–אנג'לס לניו–יורק, ומייצרת את הפלט המוצג בטבלה 9–55:

```
SELECT * FROM dbo.RoadPaths
WHERE from_city = 'LAX' AND to_city = 'JFK';
```

טבלה 9–55: מסלול קצר ביותר בין לוס–אנג'לס לניו–יורק

from_city	*to_city*	*distance*	*path*
LAX	JFK	2945	.LAX.DEN.MCI.ORD.JFK.

פתרון יעיל יותר לבעיית המסלולים הקצרים ביותר משתמש בלולאות במקום ב–CTEs רקורסיביים. הוא יעיל יותר מסיבות דומות לאלו שתיארתי קודם; כלומר, בכל איטרציה של הלולאה יש לך גישה לכל הנתונים שנאספו קודם ולא רק לרמה הקודמת המיידית. אתה יוצר פונקציה הנקראת fn_RoadsTC המחזירה משתנה טבלה הנקרא @RoadsTC. למשתנה הטבלה יש את הטורים from_city, to_city, distance ו–route, ששמותיהם מסבירים את עצמם. קוד הפונקציה ראשית מכניס לתוך @RoadsTC שורה עבור כל זוג (city1, city2) ו– (city2, city1) מטבלת Roads. הקוד אז נכנס ללולאה החוזרת על עצמה כל עוד האיטרציה הקודמת הכניסה שורות ל–@RoadsTC. בכל איטרציה של הלולאה, הקוד מכניס מסלולים חדשים המרחיבים את המסלולים הקיימים ב–@RoadsTC. מסלולים חדשים נוספים רק אם המקור והיעד אינם מופיעים כבר ב–@RoadsTC עם אותו מרחק או עם מרחק קצר יותר. הרץ את הקוד בקטע–קוד 9–43 כדי ליצור את הפונקציה fn_RoadsTC.

קטע–קוד 9–43: קוד יצירה של ה– fn_RoadsTC UDF

```
IF OBJECT_ID('dbo.fn_RoadsTC') IS NOT NULL
  DROP FUNCTION dbo.fn_RoadsTC;
GO
CREATE FUNCTION dbo.fn_RoadsTC() RETURNS @RoadsTC TABLE
(
  uniquifier INT          NOT NULL IDENTITY,
  from_city  VARCHAR(3)   NOT NULL,
  to_city    VARCHAR(3)   NOT NULL,
  distance   INT          NOT NULL,
  route      VARCHAR(MAX) NOT NULL,
  PRIMARY KEY (from_city, to_city, uniquifier)
)
```

```
AS
BEGIN
  DECLARE @added AS INT;

  INSERT INTO @RoadsTC
    SELECT city1 AS from_city, city2 AS to_city, distance,
      '.' + city1 + '.' + city2 + '.'
    FROM dbo.Roads;

  SET @added = @@rowcount;

  INSERT INTO @RoadsTC
    SELECT city2, city1, distance, '.' + city2 + '.' + city1 + '.'
    FROM dbo.Roads;

  SET @added = @added + @@rowcount;

  WHILE @added > 0 BEGIN
    INSERT INTO @RoadsTC
      SELECT DISTINCT TC.from_city, R.city2,
        TC.distance + R.distance, TC.route + city2 + '.'
      FROM @RoadsTC AS TC
        JOIN dbo.Roads AS R
          ON R.city1 = TC.to_city
      WHERE NOT EXISTS
        (SELECT * FROM @RoadsTC AS TC2
         WHERE TC2.from_city = TC.from_city
           AND TC2.to_city = R.city2
           AND TC2.distance <= TC.distance + R.distance)
        AND TC.from_city <> R.city2;

    SET @added = @@rowcount;

    INSERT INTO @RoadsTC
      SELECT DISTINCT TC.from_city, R.city1,
        TC.distance + R.distance, TC.route + city1 + '.'
      FROM @RoadsTC AS TC
        JOIN dbo.Roads AS R
          ON R.city2 = TC.to_city
      WHERE NOT EXISTS
        (SELECT * FROM @RoadsTC AS TC2
         WHERE TC2.from_city = TC.from_city
```

```
            AND TC2.to_city = R.city1
            AND TC2.distance <= TC.distance + R.distance)
        AND TC.from_city <> R.city1;

    SET @added = @added + @@rowcount;
  END
  RETURN;
END
GO
```

הפונקציה עשויה להחזיר יותר משורה אחת עבור אותן ערי מקור ויעד. כדי להחזיר את המסלולים והמרחקים הקצרים ביותר, השתמש בשאילתה הבאה:

```
SELECT from_city, to_city, distance, route
FROM (SELECT from_city, to_city, distance, route,
        RANK() OVER (PARTITION BY from_city, to_city
                     ORDER BY distance) AS rk
      FROM dbo.fn_RoadsTC()) AS RTC
WHERE rk = 1;
```

שאילתת הטבלה הנגזרת מקצה ערך דירוג (rk) לכל שורה, בהתבסס על מחיצות from_city, to_city, ולפי סדר distance. המשמעות היא שהמסלולים הקצרים ביותר יוקצו עם ערך הדירוג 1. השאילתה החיצונית מסננת רק מסלולים קצרים ביותר (rk = 1).

כשתסיים לבצע שאילתות על הטבלה RoadPaths, אל תשכח להסיר אותה:

```
DROP TABLE dbo.RoadPaths;
```

סיכום

פרק זה דן בטיפול בגרפים, עצים והיררכיות. הצגתי פתרונות איטרטיביים/רקורסיביים לגרפים, וכן פתרונות בהם אתה מוסיף מידע המתאר את העץ. היתרון העיקרי של הפתרונות האיטרטיביים/רקורסיביים הוא שאינך צריך להוסיף ולתחזק טורים נוספים; אלא, המניפולציה על הגרף מתבססת על טורי המקצוע המאוחסנים. פתרון מסלול האבות הממומש, מממש מסלול אבות, ויכול לשמור גם את הרמה עבור כל צומת בעץ. תחזוקת המידע הנוסף אינה מאוד יקרה, ובתמורה אתה מקבל שאילתות מהירות ומבוססות–סטים. פתרון הסטים המקוננים מוסיף ערכי שמאל וימין המייצגים יחסי הכלה של סטים, ויכול לשמור גם את הרמה בעץ. זהו הפתרון האלגנטי ביותר שהצגתי, והוא גם מאפשר שאילתות פשוטות ומהירות. עם זאת, תחזוקת המידע הנוסף יקרה מאוד, כך שפתרון זה מעשי עבור עצים סטטיים או עצים דינמיים קטנים.

בסעיף האחרון, הצגתי פתרונות עבור בעיות ה– transitive closure והמסלול הקצר ביותר.

מכיוון שפרק זה מסיים את הספר, אני מרגיש שעלי להוסיף מספר מילות סיכום.

אם תשאל אותי מה הדבר החשוב ביותר שאני מקווה שתיקח איתך מספר זה, אומר **מתן תשומת לב מיוחדת לעקרונות היסוד**. אל תמעיט בערכם ובחשיבותם. השקע זמן בזיהוי, התמקדות ושיפור שיטות מפתח יסודיות. כשאתה עומד בפני בעיה קשה, פתרונות יזרמו בצורה טבעית.

> "בעניינים בעלי חשיבות רבה יש לטפל בצורה קלילה."
>
> "בעניינים בעלי חשיבות נמוכה יש לטפל בצורה רצינית."
>
> — האגאקורה, ספר הסאמוראי מאת יאמאמוטו טסונטומו.

המשמעות של אימרות אלו אינה מה שנראה על פני השטח. הספר ממשיך להסביר:

> "בין ענייניו של אדם לא צריכים להיות יותר משניים או שלושה עניינים בעלי מה שאדם יוכל לכנות חשיבות מרובה. אם אלו ייבחנו בזמנים רגילים, הם יכולים להיות מובנים. חשיבה מראש על דברים ואז טיפול בהם בצורה קלילה בבוא העת, זוהי מהות העניין. לעמוד בפני אירוע ולפתור אותו בצורה קלילה הוא דבר קשה, אם לא מצאת את הפתרון בטרם עת, ותמיד תחוש חוסר ודאות בהשגת המטרה שלך. עם זאת, אם הבסיס הונח קודם לכן, תוכל לחשוב על האימרה, 'בעניינים בעלי חשיבות רבה יש לטפל בצורה קלילה', כבסיס הפעולה שלך."

א

חידות היגיון

בבסיסן של שאילתות קיימת לוגיקה. SQL היא לוגיקה, וכל בעיית שאילתות ביסודה היא חידת היגיון. החלק הקשה ביותר של פתרון בעיית שאילתה הוא בדרך כלל זיהוי ההיבטים הלוגיים שלה. תוכל לשפר את יכולת פתרון בעיות ה–SQL שלך על ידי תרגול חידות היגיון טהורות.

לפני זמן מה, הצגתי שתי חידות היגיון בטור ה–T-SQL שלי ב– SQL Server Magazine (www.sqlmag.com). ביקשתי להראות את הקשר החזק בין SQL ולוגיקה. הרעיון המקורי היה להציג את שתי החידות הללו בלבד. אך הקוראים גילו עניין כה רב בחידות – מעניין שאפילו יותר מאשר בחידות ה–T-SQL – שהחלטתי להמשיך בנוהג ולהציג חידת היגיון חדשה בכל חודש. אני רוצה להודות ל– SQL Server Magazine, אשר בנדיבותם הרשו לי לחלוק את החידות מהטור שלי עם קוראי הספר. מרבית החידות שתראו כאן לקוחות מהטור שלי.

אני רוצה גם להודות לאיתן פרחי, לובור קולר, גבריאל בן–גן, לילך בן–גן, פרננדו ג. גווררו, הדר לוין, מייקל ריס, סטיב קאס, דניס גובו, קן הנדרסון ואריק וירמן, שהציגו לי את החידות לראשונה.

חידות

סעיף זה מציג חידות היגיון. את הפתרונות לחידות תוכל למצוא בסעיף שאחריו. מישהו אמר פעם, "חידה היא התגמול של עצמה". תיהנו!

חידה 1: טבליות תרופה

תאר לך שאובחנה אצלך מחלה נדירה. הרופא רשם לך שתי תרופות – נקרא להם א' ו–ב'. כל בקבוק מכיל שלוש טבליות, ולטבליות של שתי התרופות יש בדיוק את אותו הגודל, הצורה, הצבע והריח. כל בקבוק מסומן עם סוג התרופה, בקבוק אחד ב–א' ובקבוק אחד ב–ב', אך הטבליות עצמן אינן מסומנות. ההוראות שקיבלת הן לקחת טבלית אחת מבקבוק א' וטבלית אחת מבקבוק ב' בכל יום למשך שלושה ימים. אם תמלא אחר ההוראות במדויק תבריא לחלוטין, אך אם לא תמלא אחריהן במדויק תמות מייד.

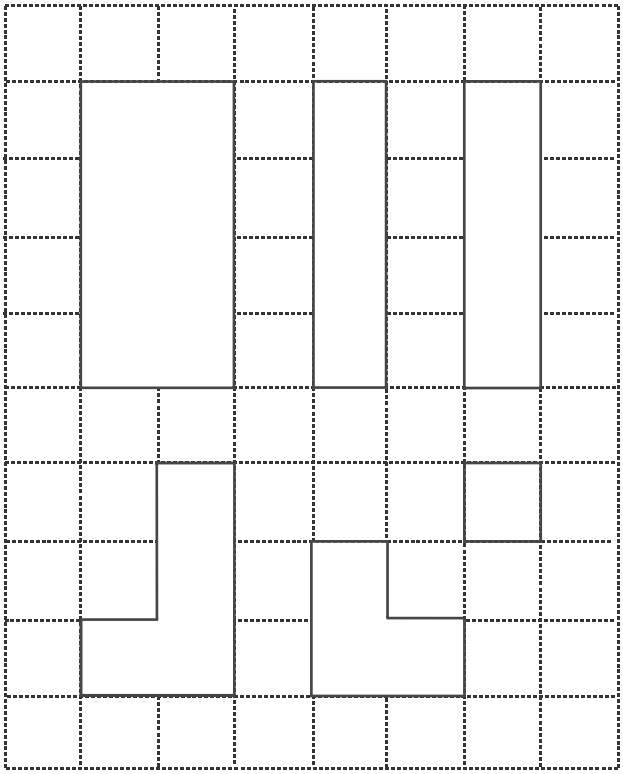
ביום הראשון, אתה לוקח טבלית אחת מבקבוק א' וטבלית אחת מבקבוק ב'. ביום הבא אתה מגלה שמישהו התעסק עם הבקבוקים שלך: בקבוק ב' ריק, בקבוק א' מכיל טבלית אחת, ושלוש טבליות מונחות על הדלפק. אתה מבין שטבלית אחת של בקבוק א' ושתי טבליות של בקבוק ב' מעורבבות כעת על הדלפק, אך אינך יכול לדעת מאיזה סוג כל טבלית. אתה מתקשר לבית המרקחת ומגלה שהטבליות אזלו מהמלאי עד מחר. כיצד תוכל להמשיך למלא אחר ההוראות במדויק כדי להירפא?

חידה 2: חפיסת שוקולד

תאר לך שבידך חפיסת שוקולד המורכבת מ–40 קוביות מסודרות ב–5 שורות ו–8 טורים. משימתך היא לחלק את החפיסה ל–40 קוביות השוקולד הבודדות תוך שימוש במספר מינימלי של חיתוכים. מותר לך לבצע חיתוך אחד בלבד בכל פעם (ואסור לך לערום מספר שכבות או להניח אותן אחת ליד השנייה) ואך ורק בקו ישר (אנכי או אופקי). מהו המספר המינימלי של חתכים לו תידרש? הוכח את ההיגיון; אל תסתפק בניחוש.

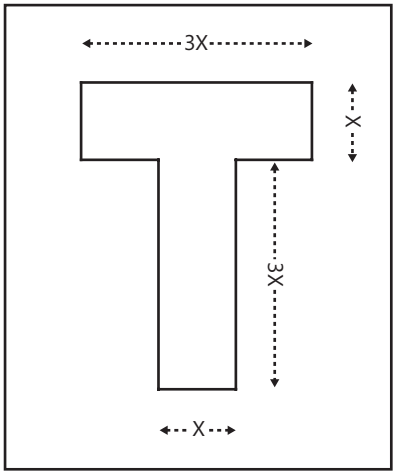
חידה 3: בניית T

העתק את הצורות שבתרשים א–1 על פיסת נייר בעלת משבצות, ואז גזור את הצורות. השתמש בצורות כדי לייצר את הצורה T בעלת הפרופורציות המוצגות בתרשים א–2. ייתכן שתצטרך לחשוב מחוץ לקופסא כדי לפתור את החידה הזו!



תרשים א–1:
חידת T – חתיכות

תרשים א–2: חידת T – תוצאה רצויה



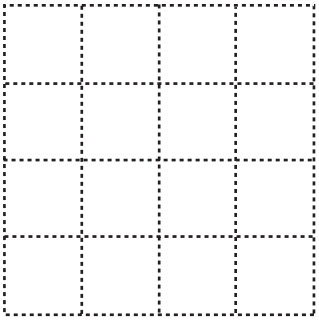
חידה 4: על הנקודה

שתי נקודות מצוינות בצורה שרירותית על פני קובייה. הגדר את הדרך הקצרה ביותר על פני הקובייה המחברת בין שתי הנקודות.

חידה 5: מלבנים בריבוע

על פיסת נייר, דמיין ריבוע בן 4 טורים ו–4 שורות, כפי שמוצג בתרשים א–3.

תרשים א–3: ריבוע 4–על–4



כמה מלבנים שונים ביכולתך לזהות על הרשת שבתוך הריבוע (כולל את הקווים החיצוניים היוצרים את הריבוע 4–על–4)?

חידה 6: מדידת זמן על ידי חבלים נשרפים

ברשותך שני חבלים. לכל חבל לוקח בדיוק שעה אחת להישרף מקצה לקצה. לחבל יש צפיפות חומר שונה בנקודות שונות, כך שאין שוויון בזמן שלוקח לחלקים שונים של החבל להישרף (למשל, החצי הראשון יכול להישרף בתוך 59 דקות והחצי השני בתוך דקה אחת). עליך למצוא דרך למדוד בדיוק 45 דקות על ידי שימוש בשני חבלים אלו בלבד (וגפרורים!). זו אינה שאלה עם טריק, כך שאיני מחפש תשובה כמו "אתה מביט בשעונך במשך 45 דקות ונהנה מהאור של החבלים הנשרפים". לחידה יש פתרון לוגי הנשען אך ורק על הבעירה של החבלים.

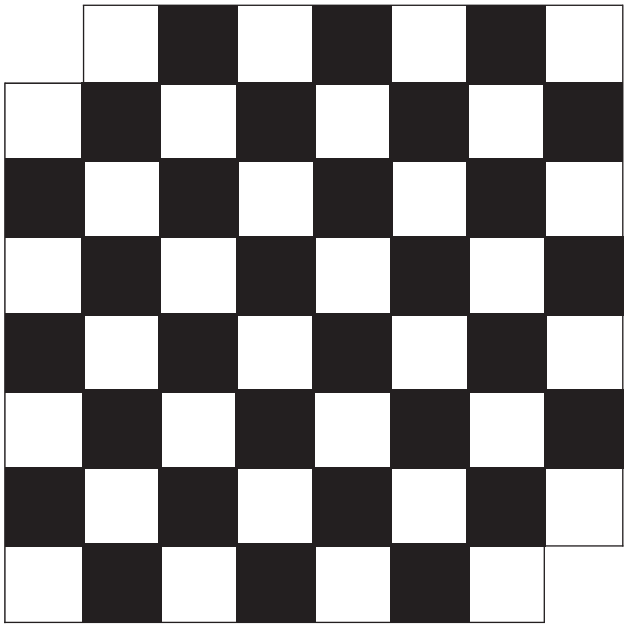
חידה 7: חישוב מקסימום אריתמטי

בהינתן שני ערכי הקלט m ו–n, חשב את הגדול מבין השניים בעזרת שימוש בביטוי מתמטי יחיד. אל תשתמש בחישוב שלך באלמנטים תכנותיים של תנאים כמו IF או CASE.

חידה 8: כיסוי לוח שחמט עם לבני דומינו

ברשותך לוח שחמט עם שתי פינות חסרות, כפי שמוצג בתרשים א–4.

תרשים א–4: לוח שחמט עם שתי פינות חסרות



ברשותך 31 לבני דומינו, כל אחת בגודל של שתי משבצות בדיוק. האם באפשרותך לכסות את כל 62 המשבצות הנותרות של לוח השחמט בלבני דומינו מבלי לכסות את הפינות החסרות? אם התשובה היא כן, הצע סידור ללבני הדומינו. אם התשובה היא לא, הוכח לוגית שהדבר אינו אפשרי. שים לב שלבני הדומינו אינן יכולות לבלוט מחוץ ללוח השחמט, אינן יכולות להיות מונחות זו על גבי זו, ואסור לך לשבור אותן.

חידה 9: השקל החסר

שלושה אנשים מגיעים למלון ומבקשים לחלוק חדר. המחיר הוא 30 ₪, כך שכל אחד נותן 10 ₪. מאוחר יותר, פקיד הקבלה מגלה שהוא חייב את האורחים חיוב יתר של 5 ₪. מכיוון שאינו יכול לחלק 5 ₪ בצורה שווה בין שלושת האורחים, הוא מכניס לכיסו 2 ₪ ומחזיר לכל לקוח שקל אחד בחזרה. בלילה מתקשה פקיד הקבלה לישון מכיוון שנראה כי המספרים לא מתחברים. כל לקוח שילם לבסוף 9 ₪, ו– 29 ₪ = 2 ₪ + 3 X 9 ₪. היכן השקל החסר?

חידה 10: כיבוי והדלקה של מפסקי מנורה

מאה מנורות מסודרות בשורה (נקרא להם מנורה 1, מנורה 2, מנורה 3, וכך הלאה). במצב ההתחלתי כל המנורות כבויות, ולכל אחת יש מפסק כיבוי/הדלקה. מאה אנשים – איש 1, איש 2, איש 3, וכך הלאה – ממלאים את המשימה הבאה: שנה את מצב המפסק של מנורה 1n, מנורה 2n, מנורה 3n, וכך הלאה כאשר n הוא מספר האיש. כך שאיש 1 משנה את המצב של מפסקי מנורות 1, 2, 3, וכך הלאה; איש 2 משנה את מצב מפסקי מנורות 2, 4, 6, וכך הלאה; ובצורה דומה אנשים 3 עד 100 משנים את מצב המפסקים לפי ההוראות. אילו מנורות נשארות מוארות לאחר שכל מאה האנשים סיימו את משימתם?

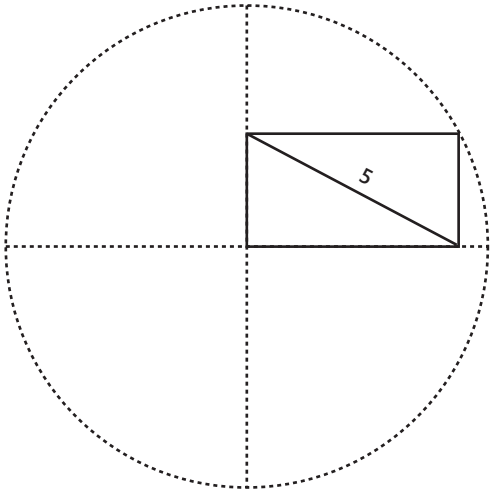
חידה 11: חיתוך מקל ליצירת משולש

חידה זו מערבת חישובי הסתברות. נניח שחתכת מקל בשתי נקודות שרירותיות. מהי ההסתברות שתוכל ליצור משולש משלוש החתיכות?

חידה 12: מלבן בתוך מעגל

חידת היגיון זו עוסקת בגיאומטריה. בחן את תרשים א–5. האם תוכל לחשב את האורך של רדיוס המעגל?

תרשים א–5: מלבן בתוך מעגל



חידה 13: מה מאחורי הווילון?

אתה לוקח חלק בשעשועון. לפניך שלושה וילונות. מאחורי אחד מהם מכונית. אינך יודע מי וילון הפרס, אך מנחה השעשועון יודע.

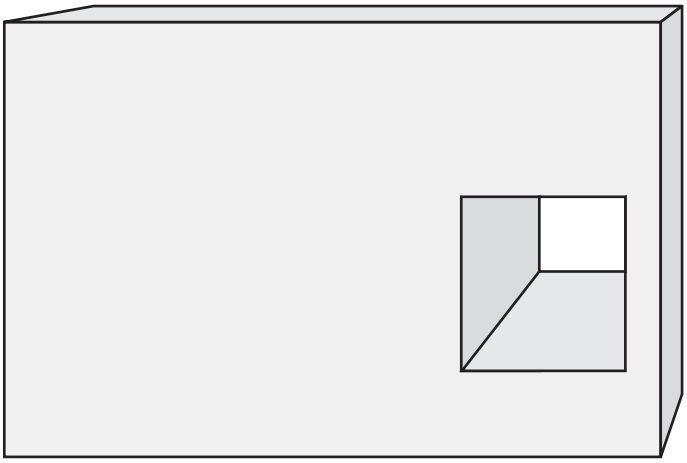
משימתך היא לנחש איזה וילון מסתיר את המכונית. אם תבחר בווילון שמאחוריו המכונית, המכונית שלך. לאחר שבחרת וילון, המנחה פותח את אחד משני הווילונות האחרים, שמאחוריו ידוע לו שאין מכונית. המנחה אז נותן לך הזדמנות לשנות את דעתך בנוגע למיקום המכונית. האם עליך להיצמד לבחירה המקורית שלך, או שעליך לבחור בווילון הסגור השני?

חידה 14: עוגה עם חור

ברשותך עוגה מלבנית ובה חור מלבני במקום כלשהו. על העוגה מרוחה שכבת קרם (אך לא בצידיה). תרשים א–6 מציג עוגה כזו.

ברשותך גם סכין ועליך לחתוך את העוגה לשתי מנות שוות – שוות מבחינת כמות עוגה וכמות קרם. כיצד תוכל למלא משימה זו בעזרת חיתוך ישר יחיד?

תרשים א–6: עוגה עם חור



חידה 15: קלפים פונים כלפי מעלה

ברשותך חבילה של 52 קלפי משחק, כאשר בתוך החבילה 7 קלפים שפניהם כלפי מעלה, והיתר פניהם מטה. אינך יודע איזה מהקלפים פניהם כלפי מעלה ואיזה כלפי מטה, ואינך יכול להתבונן בקלפים כדי לגלות מכיוון שעיניך מכוסות.

המשימה שלך היא להפריד את הקלפים לשתי ערימות, כך שכל ערימה תכיל את אותה כמות של קלפים שפניהם כלפי מעלה. זכור, עיניך מכוסות. כיצד תוכל למלא את המשימה?

הנה רמז: מספר הקלפים בכל ערימה אינו חייב להיות זהה, רק מספר הקלפים שפניהם כלפי מעלה. כמו כן, המספר המדויק של קלפים שפניהם כלפי מעלה בכל ערימה אינו חשוב בפני עצמו; מה שחשוב הוא שיהיה מספר זהה של קלפים שפניהם מעלה בכל ערימה.

חידה 16: חשבון בסיסי

האם באפשרותך לשלב את השלמים 3, 4, 5, ו–6 להשגת 28, תוך שימוש אך ורק בסימני החישוב +, –, ×, ו– ÷ וכן סוגר ימני ושמאלי? מותר לך להשתמש בכל מספר פעם אחת בלבד ובכל סימן חישוב וסוגר פעם אחת בלבד.

חידה 17: קוד שמשכפל את עצמו (Quine)

חידה זו, אותה מצאתי מאוד מעניינת ומאתגרת, הוצגה על ידי קן הנדרסון בבלוג שלו (http://blogs.msdn.com/khen1234/archive/2005/10/25/484555.aspx).

האתגר העומד בפניך הוא לכתוב קוד quine (קוד שמשכפל את עצמו) ב–T-SQL. כלומר, עליך לכתוב קוד המייצר פלט הזהה לקוד שייצר אותו. אינך מורשה לבצע שאילתות על metadata או מבנים פנימיים המעניקים לך גישה לאזור החיץ של הקוד של ה–session או למלל של הקוד המאוחסן. אלא, עליך להישען על מניפולציה תקנית של SQL. למשל, הפתרון הבא אינו מותר, על אף שהוא משכפל את עצמו:

```
DECLARE @sql AS NVARCHAR(MAX);
SET @sql = (SELECT text FROM fn_get_sql(
   (SELECT sql_handle FROM sys.sysprocesses WHERE spid = @@spid)));
PRINT @sql;
```

מציאת פתרון תקני כלשהו היא אתגר בפני עצמה, אך ככל שהקוד קצר יותר, תקבל יותר נקודות.

חידה זו אינה מוגבלת ל–T-SQL כמובן. תוכל לנסות לפתור אותה בכל שפת תכנות. תוכל למצוא דוגמאות כאן: http://www.madore.org/~david/computers/quine.html.

חידה 18: טיול בהר

תרמילאי מטייל במעלה ההר, כשהוא מתחיל בתחתית בדיוק בזריחה ומגיע לפסגה בדיוק בשקיעה. ביום הבא התרמילאי יורד מההר, כשהוא מתחיל בפסגה בדיוק בזריחה ומגיע לתחתית בדיוק בשקיעה, ומשתמש בדיוק באותו מסלול בו הלך ביום הקודם. הנח שלא היה כל שינוי בזמני הזריחה והשקיעה. האם תוכל להוכיח שהייתה נקודה כלשהי במהלך המסלול שבה ביקר התרמילאי בדיוק באותה השעה בשני הימים?

חידה 19: מצא את דפוס הסדרה

ניתנת לך תחילתה של סדרת מספרים:

3, 3, 5, 4, 4, 3, 5, 5, 4, 3, 6, 6, 8, 8

האם תוכל לזהות את הדפוס בסדרה ולמצוא כיצד תמשיך?

פתרונות לחידות

סעיף זה כולל את הפתרונות לחידות ההיגיון.

חידה 1: טבליות תרופה

חתוך כל אחת מהטבליות הנותרות לחצי. טול מחצית אחת מכל טבלית היום ואת היתר מחר.

פעם פרסמתי את החידה הזו בפורום של מדריכים. אחד מהפתרונות, שנשלח על ידי מארי בריי מאוסטרליה, היה משעשע במיוחד. מארי הציעה שתמזוג לעצמך כוס נכבדה של ויסקי, תמיס בתוכה את כל הטבליות ותשתה חצי כוס היום וחצי כוס מחר. כך, אם תמות, לפחות תמות שמח.

חידה 2: חפיסת שוקולד

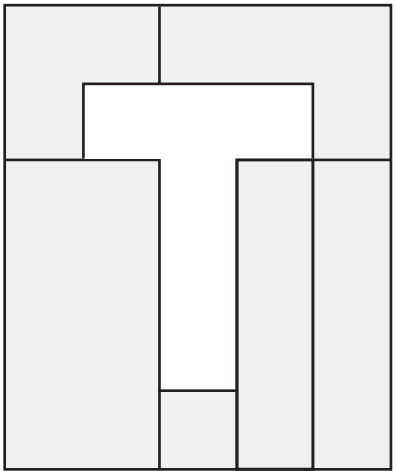
מרבית האנשים מנסים להריץ מספר תסריטים שונים – למשל, ראשית לחתוך לאורך כל קו אופקי, ואז לחתוך כל שורה אנכית. זה מבלבל להריץ את התסריטים השונים בראש, כך שקל מאוד לעשות טעויות בחישובים ולהגיע לתוצאות שונות עבור התסריטים השונים. התוצאות השונות שאתה מקבל יוצרות אשליה שקיימות מספר אפשרויות וששיטה אחת דורשת פחות חיתוכים מאחרת. אך האמת היא, שבכל שיטה בה תשתמש תמיד תבצע 39 חיתוכים.

בתחילה, ברשותך חתיכה אחת של שוקולד. כל חיתוך שתבצע תהפוך חתיכה אחת (החתיכה אותה אתה מחלק) לשתיים, במילים אחרות, כל חיתוך שתבצע יגדיל את מספר החתיכות שברשותך בדיוק באחד. לאחר החיתוך הראשון, ברשותך שתי חתיכות. לאחר החיתוך השני, ברשותך שלוש חתיכות. יהיו ברשותך 40 החתיכות המבוקשות לאחר 39 חיתוכים בדיוק.

חידה 3: בניית T

מרבית האנשים מנסים את הגישה המובנת מאליה – יצירת הצורה T עצמה (בהדגשה על עצמה) על ידי סידור החתיכות בצורות שונות. אם ניסית גישה זו, סביר להניח שלא מצאת פתרון, ומסיבה טובה – אינך יכול ליצור את הצורה T בפרופורציות המבוקשות על ידי אף סידור של החתיכות הנתונות. אך אם תחשוב מחוץ לקופסה (מילולית במקרה זה), תוכל ליצור את המרובע שמסביב לצורה T, כפי שמציג תרשים א–7.

תרשים א–7: פתרון לחידת T



חידה 4: על הנקודה

דרך טובה לגשת לבעיה זו היא להתחיל במקרה הפשוט – כאשר שתי הנקודות נמצאות על אותה פאה – ואז לנסות להכליל למקרים מורכבים יותר. כמובן שקו ישר הוא המסלול הקצר ביותר המחבר שתי נקודות המסומנות על אותה פאה. כדי להפעיל את אותו היגיון למקרים המורכבים יותר בהם הנקודות מסומנות על פאות שונות, עליך לדמיין את פני הקובייה כמישור על ידי פריסה והשטחה של הפאות הסמוכות. לאחר שברשותך מישור שטוח, המסלול הקצר ביותר בין הנקודות הוא הקו הישר המחבר אותן.

קיימות מספר דרכים לשטח את פני הקובייה ליצירת מישור. הקו הישר הוא המסלול הקצר ביותר בין שתי הנקודות על המישור של הפאות הפרוסות, אך הוא אינו תמיד המסלול הקצר ביותר על הקובייה, והוא גם לא תמיד מוכל בפני השטחים הפרוסים. (ייתכן שתרצה "לחתוך" את פני השטח של הקובייה דרך קצה על המסלול הקצר האמיתי, שיאלץ אותך "לקפוץ" לפאה הסמוכה או "ללכת בדרך הארוכה"). מטרת החידה הייתה לזהות את הדרך הלוגית שהיא לשטח את פאות הקובייה ליצירת משטח מישורי ומתיחת קו ישר בין נקודות. כך שאינך צריך ממש להגדיר איזו מהפריסות השונות תיתן לך את הקו הישר הקצר ביותר. מספיק לומר שהמסלול הקצר ביותר על פני הקובייה בין שתי נקודות הוא הקצר מבין הקווים הישרים המחברים את הנקודות מבין הפריסות השונות של פאות הקובייה.

חידה 5: מלבנים בריבוע

כדי לפתור את החידה הזו, ייתכן שתתפתה פשוט לספור את כל המלבנים השונים שניתן לצייר בתוך ריבוע. אלא ששיטה זו מבלבלת ובדרך כלל מובילה לתוצאה שגויה. אני נהנה מחידת הקומבינטוריקה הזו מכיוון שהפתרון טמון בגישה שיטתית.

ראשית, קבע אילו סוגי מלבנים אפשריים; שנית, קבע כמה מלבנים יש מכל סוג; שלישית, חבר הכל. הסוגים השונים של מלבנים שביכולתך לשרטט בריבוע 4–על–4 הם מלבנים להם בין שורה לארבע שורות ובין עמודה לארבע עמודות. תוכל להשתמש בטבלה א–1 כמקום לכתוב את מספר המלבנים בכל סוג אפשרי.

טבלה א–1: מספר מלבנים בכל סוג אפשרי

Rows/Columns	*1*	*2*	*3*	*4*
1				
2				
3				
4				

מלא כל תא עם מספר המלבנים של הגודל הספציפי (rows (r) x columns (c)) שביכולתך לשרטט בריבוע 4–על–4. את מספר המלבנים השונים בגודל r x c שביכולתך לשרטט בריבוע של n x n ניתן לבטא כ– $(n - r + 1) \times (n - c + 1)$. ההיגיון מאחורי נוסחה זו היא שבתוך n שורות, קיימים $n - r + 1$ מיקומים אנכיים שונים שבהם תוכל להציב את השורה העליונה של מלבנים בעלי r שורות. באופן דומה, קיימים $n - c + 1$ מיקומים אופקיים אפשריים שבהם תוכל להציב את העמודה השמאלית של מלבן בעל c עמודות. כך שבסך הכל, אנחנו מדברים על $(n - r + 1) \times (n - c + 1)$ מלבנים שונים בגודל r x c. כאשר תמלא את כל התאים במטריצה בהתאם לנוסחה זו, תקבל את המטריצה המוצגת בטבלה א–2.

טבלה א–2: מטריצה מלאה בערכי שורה ועמודה

Rows/Columns	*1*	*2*	*3*	*4*
1	4 × 4	4 × 3	4 × 2	4 × 1
2	3 × 4	3 × 3	3 × 2	3 × 1
3	2 × 4	2 × 3	2 × 2	2 × 1
4	1 × 4	1 × 3	1 × 2	1 × 1

כעת פשוט חבר את הערכים ותקבל את התשובה לחידה:

```
4 × 4 + 4 × 3 + 4 × 2 + 4 × 1 +
3 × 4 + 3 × 3 + 3 × 2 + 3 × 1 +
2 × 4 + 2 × 3 + 2 × 2 + 2 × 1 +
1 × 4 + 1 × 3 + 1 × 2 + 1 × 1 =

4 × (4 + 3 + 2 + 1) +
3 × (4 + 3 + 2 + 1) +
2 × (4 + 3 + 2 + 1) +
1 × (4 + 3 + 2 + 1) =

(4 + 3 + 2 + 1) × (4 + 3 + 2 + 1) =

(1 + 2 + 3 + 4)² = 100
```

הפתרון למקרה כללי יותר של ריבוע n x n הוא $(1 + 2 + 3 + \ldots + n)^2$.

קיימת גישה אחרת, אולי פשוטה יותר, שהוצעה על ידי סטיב קאס. בכל תא 1–על–1 של הריבוע המקורי 4–על–4 של החידה, רשום את מספר המלבנים אשר הפינה ה"דרום–מערבית" שלהם היא בנקודה זו. מספר זה קל למציאה – זהו המספר של הפינות ה"צפון–מזרחיות" האפשריות, או מספר התאים שאינם מתחת לתא הנתון ואינם משמאלו. מתקבל המערך הבא של ספירות:

```
 4,    3,    2,    1
 8,    6,    4,    2
12,    9,    6,    3
16,    12,   8,    4
```

להשגת הפתרון הכללי, שים לב שהשורה העליונה מסתכמת ל– $1 \times (1 + 2 + 3 + \ldots + n)$, השורה שמתחתיה מסתכמת ל– $2 \times (1 + 2 + 3 = \ldots + n)$, וכך הלאה, כך שהתוצאה היא $(1 + 2 + 3 + \ldots + n) \times (1 + 2 + 3 + \ldots + n)$. ומתקבלת התוצאה המופשטת $(1 + 2 + 3 + \ldots + n)^2$.

חידה 6: מדידת זמן על ידי חבלים נשרפים

הדלק את שני הקצוות של חבל אחד, וקצה אחד של החבל השני. ברגע שהחבל הראשון מתכלה (מה שייקח בדיוק 30 דקות), הדלק את הקצה הכבוי של החבל השני. החבל השני יתכלה בדיוק לאחר 15 דקות נוספות, כך שמדדת בסך הכל בדיוק 45 דקות.

חידה 7: חישוב מקסימום אריתמטי

הפתרון הוא: $(m + n)/2 + abs(m - n)/2$. אם תחשוב על כך, הערך הגדול מבין שני ערכים הוא הממוצע שלהם ועוד מחצית מהערך המוחלט של ההפרש ביניהם. הממוצע של שני ערכים שווה לערך הקטן ועוד מחצית ההפרש ביניהם. כך שהממוצע ועוד מחצית ההפרש שווה לקטן ועוד פעמיים מחצית ההפרש, שהוא כמובן הערך הגדול מבין השניים. ייתכן שבמילים בלבד קצת קשה לעקוב אחר הפתרון. דרך אחרת להצדיק את התשובה היא להראות שהיא נכונה בין אם [מקרה 1] $m >= n$ (שבו abs(m-n) הוא פשוט m-n, וההסבר מופשט לכדי $((m + n)/2 + (m - n)/2 = 2m/2 = m$) ובין אם [מקרה 2] $m < n$ (שבו $abs(m-n) = n-m$ וההסבר מופשט לכדי $((m + n)/2 + (n – m)/2 = 2n/2=n)$. מכיוון שאחד מהמקרים 1 או 2 תמיד נכון, הנוסחה תמיד תעבוד.

חידה 8: כיסוי לוח שחמט עם לבני דומינו

כדי לפתור חידה זו, ספור את המשבצות הלבנות והשחורות. שתי הפינות החסרות הן בעלות צבע זהה – שתיהן לבנות או ששתיהן שחורות – מה שמשאיר 32 משבצות מצבע אחד ו–30 מהצבע השני. כל לבנת דומינו מכסה בדיוק שתי משבצות – אחת שחורה ואחת לבנה. כתוצאה מכך, כל סידור של הדומינו יכסה את אותו מספר משבצות מכל צבע. לפיכך, בלתי אפשרי לכסות לחלוטין את כל 62 המשבצות הנותרות של לוח השחמט עם לבני דומינו.

חידה 9: השקל החסר

להלן החלוקה של 30 השקלים: המלון קיבל 25 ₪, פקיד הקבלה קיבל 2 ₪, ולאורחים יש 3 ₪. השגיאה בחישוב של הפקיד הייתה שהוא חיבר 27 ₪ ועוד 2 ₪ – אך הסיבה לכך שזו שגיאה היא שאין כל היגיון בחיבור המספרים הללו מכיוון ש–2 השקלים (שהוא שם בכיס) היו כבר חלק מ–27 השקלים ששילמו האורחים. 30 השקלים המקוריים, אם עליך לוודא היכן נמצאים, הם 27 השקלים ששילמו האורחים ועוד 3 השקלים שקיבלו בחזרה.

חידה 10: כיבוי והדלקה של מפסקי מנורה

כל המנורות כבויות מלבד מנורות 1, 4, 9, 16, 25, 36, 49, 64, 81, ו–100 שהן דלוקות. הפתרון לחידה זו מתמטי בבסיסו.

מספר הפעמים שהמפסק של מנורה n שינה מצב, שווה למספר המחלקים השלמים של n. המצב הסופי של המנורה תלוי אך ורק בשאלה האם מספר המחלקים השלמים הוא זוגי או אי–זוגי.

המקרים היחידים שבהם ל–n יש מספר אי–זוגי של מחלקים הם כאשר n הוא מספר ריבועי. הסיבה לכך היא שכל מחלק k הקטן מהשורש הריבועי של n יכול להיות הדו של n/k, השונה מ–k. לפיכך, כל המנורות במיקומים שלהם שורש ריבועי שלם יסיימו את התהליך במצב הפוך למצב ההתחלתי שלהן, בעוד שכל המנורות האחרות יישארו במצבן המקורי.

חידה 11: חיתוך מקל ליצירת משולש

כדי ששלושת החלקים ירכיבו משולש, כל החלקים חייבים להיות קצרים ממחצית האורך של המקל המקורי, מכיוון שבמשולש סכום האורכים של כל זוג צלעות תמיד גדול מהאורך של הצלע הנותרת. אם חלק אחד גדול ממחצית גודל המקל, סכום אורך שני החלקים האחרים יהיה קטן יותר ולא תוכל להרכיב משולש.

נניח שהחיתוך הראשון הוא ב–p%. אם p הוא בין 0 ל–50%, ניתן להרכיב משולש רק אם החיתוך השני הוא מעל 50% (החיתוך השני חייב להיות בצד השני של מרכז המקל) ומתחת ל– p + 50% (כל אחד משני החיתוכים חייב להיות בתוך מחצית האורך המקורי של השני כדי להימנע מחתיכה ארוכה מדי). במילים אחרות, החיתוך השני חייב להיות בתוך החלק מ–50% ל– p + 50%. חלק זה הוא באורך p, כך שההסתברות שהחיתוך השני יאפשר משולש היא p במקרה זה.

אם החיתוך הראשון הוא בין 50% ל–100%, החיתוך השני (כדי לאפשר משולש) חייב להיות לכל הפחות p – 50% ולכל היותר 50%, או אינטרוול של אורך 50% - (p – 50%) = 1 – p. כך שסך ההסתברות שניתן יהיה להרכיב משולש היא:

$$\int_{0}^{0.5} p\,dp + \int_{0.5}^{1} (1-p)\,dp = \frac{1}{4}$$

תוכל למצוא מספר נוסחאות לפתרון כאן:

.http://www.cut-the-knot.org/Curriculum/Probability/TriProbability.shtml#Explanation

חידה 12: מלבן בתוך מעגל

ברצוננו לחשב את רדיוס המעגל (כפי שמוצג בתרשים א–5). התכסיס פה הוא לזהות שאלכסון המלבן שאינו משורטט הוא רדיוס המעגל. מכיוון ששני האלכסונים במלבן הם בעלי אורך זהה, הרדיוס הוא 5.

כאשר מנסים לפתור בעיות המערבות משולשים ישרי זווית, מרבית האנשים מניחים שהפתרון מבוסס על משפט פיתגורס, ומתחילים לחשב ריבועים ושורשים מרובעים, שבמקרה זה מובילים לדרך ללא מוצא. גם כשפותרים בעיות T-SQL קל להתבלבל על ידי קביעת הנחות מהירות. עדיף להאט ולנקות את הראש. במקרים רבים, הפתרון אותו אתה מחפש פשוט הרבה יותר מאשר חשבת בהתבוננות ראשונית.

חידה 13: מה מאחורי הווילון?

מרבית האנשים חושבים שלא קיים הבדל הסתברותי בין היצמדות לבחירה המקורית שלך לבין שינוי הבחירה. במילים אחרות, הם חושבים שיש 50 אחוז הסתברות של זכייה אם תיצמד לבחירה המקורית ו–50 אחוז אם תשנה את הבחירה. אף על פי כן, זכור שהבחירה הראשונה שלך הייתה עצמאית לחלוטין (התקבלה שרירותית ללא כל ידע מוקדם), ושללא קשר לבחירה המקורית שלך, המנחה תמיד יפתח את אחד מהוילונות האחרים שהוא ריק. כך שאם תיצמד לבחירה המקורית שלך, ההסתברות לזכייה נשארת זהה ללא תלות באירוע הבא – פתיחת הווילון הריק על ידי המנחה. זה יהיה במקרה אם תבחר את וילון 1 פעמיים, וילון 2 פעמיים או וילון 3 פעמיים. בהגדרה, אם תידבק בבחירתך המקורית, יש לך סיכוי זכייה של 1/3. מצד שני, הבחירה של המנחה באיזה וילון לפתוח, תלויה בבחירה המקורית שלך ובתורו, שינוי הבחירה המקורית שלך תלוי בווילון שנפתח. לפיכך, שינוי בחירתך המקורית יהיה מבוסס על ידע חדש. כשברשותך רק שתי אפשרויות לאחר שהמנחה פתח את אחד הוילונות, הסבירויות לזכייה אם תדבק בבחירתך המקורית או תשנה את דעתך חייבות להסתכם ל–1 (100 אחוז). לפיכך, ההסתברות לזכייה אם תשנה את בחירתך היא $1 - 1/3 = 2/3$. סטטיסטית שני אנשים משלושה המחליטים לשנות את דעתם זוכים, בעוד שרק אדם אחד משלושה המחליטים לדבוק בבחירתם המקורית זוכה.

כך שעדיף לשנות את בחירתך!

אם אינך משוכנע עדיין, ואני חייב לומר שבחידה זו הרבה אנשים לא מקבלים את האמת אפילו לאחר ששמעו את ההסבר, אני יכול להציע אחד משני דברים:

1. דבר עם סטטיסטיקאים ומתמטיקאים.
2. כתוב תוכנית המדמה את השעשועון ובודקת את התוצאות סטטיסטית. להלן דוגמה לתוכנית כזו שנכתבה ב–T-SQL על ידי סטיב קאס:

```
-- Simulating the "Monty Hall Problem" in T-SQL (2005)
--   A description of the problem can be found at
--   http://math.ucsd.edu/~crypto/Monty/montybg.html)
 WITH T0 AS
 (
   SELECT
     -- prize_door is door 1, 2, or 3 with equal probability
     1 + ABS(BINARY_CHECKSUM(NEWID())) % 3 AS prize_door
   FROM dbo.Nums
   WHERE n <= 100000 -- number of trials
   -- use any handy table that is not too small
 ),
 T1 AS
 (
   SELECT
     prize_door,
     -- your_door is door 1, 2, or 3 with equal probability
     1 + ABS(BINARY_CHECKSUM(NEWID())) % 3 AS your_door
   FROM T0
 ),
 T2 AS
 (
   SELECT
   -- The host opens a door you did not choose,
   -- and which he knows is not the prize door.
   -- If he has two choices, each is equally likely.
     prize_door,
     your_door,
     CASE
       WHEN prize_door <> your_door THEN 6 - prize_door - your_door
       ELSE SUBSTRING(
                 REPLACE('123',RIGHT(your_door, 1), ''),
                 1 + ABS(BINARY_CHECKSUM(NEWID())) % 2,
                 1)
     END AS open_door
   FROM T1
 ),
 T3 AS
 (
   SELECT
     prize_door,
     your_door,
```

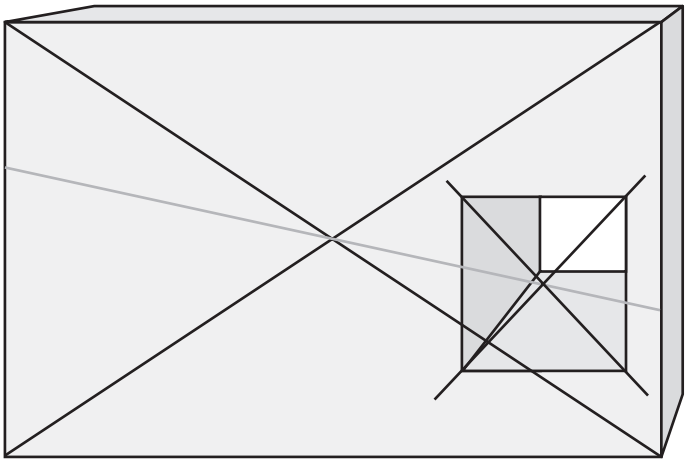
```
    open_door,
    -- The "other door" is the still-closed door
    -- you did not originally choose.
    6 - your_door - open_door AS other_door
  FROM T2
),
T4 AS
(
  SELECT
    COUNT(CASE WHEN prize_door = your_door
                THEN 'Don''t Switch' END) AS staying_wins,
    COUNT(CASE WHEN prize_door = other_door
                THEN 'Do Switch!'    END) AS switching_wins,
    COUNT(*)                                AS trials
  FROM T3
)
SELECT
  trials,
  CAST(100.0 * staying_wins / trials
       AS DECIMAL(5,2)) AS staying_winsPercent,
  CAST(100.0 * switching_wins / trials
       AS DECIMAL(5,2)) as switching_winsPercent
FROM T4;
```

בתוך הקוד ישנן הערות המסבירות את הפתרון. טבלת העזר Nums נידונה בפרק 4. אך כפי שההערות בקוד מסבירות, באפשרותך לבצע שאילתה על כל טבלה בעלת מספר שורות מספק – שהוא מספר הניסיונות שפתרון זה יבצע.

חידה 14: עוגה עם חור

כל קו ישר שעובר דרך מרכז מלבן מחלק אותו לשני חצאים בעלי גודל שווה, ללא תלות בזווית. ראשית, סמן את מרכז העוגה ואת מרכז החור, תוך התבססות על העובדה שמרכז מלבן הוא ההצטלבות של אלכסוניו. כעת בצע חיתוך ישר יחידי דרך שני המרכזים – של העוגה ושל החור, כפי שמוצג בתרשים א–8.

תרשים א–8: פתרון לחידת העוגה



חיתוך זה מחלק את העוגה ואת החור לשני חלקים שווים. מתקבלות שתי חתיכות בעלות גודל זהה וכמות זהה של קרם.

חידה 15: קלפים פונים כלפי מעלה

חידה זו היא מתמטית מטבעה. חלק את החפיסה לשתי ערימות: ערימה אחת בת 7 קלפים, וערימה אחת עם 45 הקלפים הנותרים. בשלב זה, הערימה בת 7 הקלפים מכילה מספר כלשהו (ייתכן שאפס) קלפים שפניהם כלפי מעלה. נקרא למספר זה n. מכיוון שבהתחלה היה סך של 7 קלפים שפניהם כלפי מעלה בכל החפיסה, הערימה בת 45 הקלפים חייבת להכיל 7-n קלפים שפניהם כלפי מעלה. כעת הפוך את כל הערימה בת 7 הקלפים על צידה השני. סיימת. בטרם הפכת את הערימה בת 7 הקלפים, n מתוך 7 קלפים פנו כלפי מעלה, והיתר (7-n מהם) פנו כלפי מטה. כעת כשהפכת אותם על צידם השני, המספרים התהפכו, והערימה בת 7 הקלפים מכילה 7-n קלפים שפניהם כלפי מעלה, אותו מספר שפונה מעלה כמו בערימה בת 45 הקלפים.

חידה 16: חשבון בסיסי

הרעיון כאן הוא לייצר 7 מתוך 3, 5, ו–6 ואז להכפיל אותו ב–4, מכיוון ש–28 (התוצאה המבוקשת) הוא 4 x 7. חידה זו מבלבלת מכיוון שהדרכים הקלות להשיג 7 משתמשות במספר 4 (4+3, 6–3+4, 4–6+5 וכד'). ברגע שאתה מבין שעליך ליצר 7 ולהכפיל אותו ב–4, ייתכן שתגיע לביטוי הבא: (4+3) x (2-6). אך פתרון זה אינו תקין מכיוון שהוא משתמש בסוגריים פעמיים. פתרון תקין הוא: (3 / 6 + 5) x 4.

חידה 17: קוד שמשכפל את עצמו (Quine)

לחידה זו פתרונות שונים, כולם מאתגרים למדי. להלן פתרון אפשרי אחד עליו חשבתי (שים לב שהקוד בדפוס מתפצל, על אף שהוא אמור להופיע בשורה יחידה:

```
PRINT REPLACE(SPACE(1)+CHAR(39)+SPACE(1)+CHAR(39)+CHAR(41),SPACE(1),
'PRINT REPLACE(SPACE(1)+CHAR(39)+SPACE(1)+CHAR(39)+CHAR(41),SPACE(1),')
```

סטיב קאס הצליח לצמצם אותו אף יותר על ידי שימוש בערכים בינאריים במקום הפונקציות SPACE ו–CHAR:

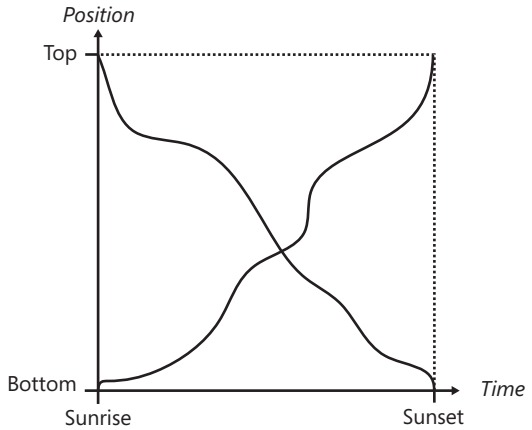
```
PRINT REPLACE(0x2027202729,0X20,'PRINT REPLACE(0x2027202729,0x20,')
```

פתרון אחר עליו חשבתי – קטע–קוד ריק. ייתכן שתחשוב שתקיפות הפתרון מוטלת בספק, אך טכנית הוא עונה על דרישות החידה: קטע–קוד ריק היא תוכנית ניתנת להרצה ב–T-SQL, והיא מייצרת תוצאה ריקה.

חידה 18: טיול בהר

ניתן להציע שתי דרכים לגשת לבעיה. דרך אחת היא לחשוב על שני תרמילאים נפרדים – אחד מטפס במעלה ההר ואחד יורד למטה – שניהם מתחילים את הטיול באותו יום בזריחה ומסיימים בשקיעה. כמובן שהם חייבים להיפגש בנקודה כלשהי. דרך אחרת היא לפתור את החידה בצורה גרפית. שרטט גרף שבו ציר X מייצג זמן (מזריחה עד שקיעה) וציר Y מייצג מיקום (מתחתית ההר עד לפסגה). שרטט שני קווים המייצגים מיקום וזמן של כל טיול. כמובן ששני הקווים חייבים להיפגש בנקודה מסוימת. דוגמה לגרף כזה מוצגת בתרשים א–9.

תרשים א–9: טיול בהר



חידה 19: מצא את דפוס הסדרה

כל ערך Vn בסדרה מייצג את מספר האותיות בשם האנגלי של n (ללא רווחים או מקפים):

one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen…

להלן החלק הראשון של הסדרה עם כמה מספרים נוספים:

3, 3, 5, 4, 4, 3, 5, 5, 4, 3, 6, 6, 8, 8, 7, 7, 9, 8, 8, 6, 9, 9, 11, 10, 10, 9, 11, 11, 10, …

סיכום

אני מקווה שאתה מוצא שחידות היגיון מאתגרות, מהנות ומשמשות כלי נהדר לשיפור הלוגיקה וה–SQL שלך. ואם אתה עדיין מחפש סיבה לתרגל אותן, הנה אחת:

"פשע הוא שכיח. היגיון הוא נדיר.

לפיכך בהיגיון ולא בפשע עליך לעסוק".

סר ארתור קונן דויל, 1859–1930, הרפתקאות שרלוק הולמס,

"The Adventure of the Copper Beeches".

על הסופרים

איציק בן–גן

איציק בן–גן הוא מנטור ואחד המייסדים של חברת Solid Quality Learning.

Microsoft SQL Server MVP (Most valuable Professional) מאז 1999, איציק העביר אירועי הדרכה רבים ברחבי העולם תוך שהוא מתמקד בשאילתות T-SQL, כוונון שאילתות ותכנות. איציק כתב מספר ספרים על Microsoft SQL Server. הוא כתב מאמרים רבים עבור SQL Server Magazine, כמו גם מאמרים ו–whitepapers עבור MSDN. בין האירועים בהם איציק הרצה ניתן למצוא את Tech Ed, DevWeek, קבוצות משתמשים שונות של SQL מסביב לעולם, PASS, SQL Server Magazine Connections ואירועי Solid Quality Learning.

מאז 1992, איציק היה מעורב בפרויקטים רבים העוסקים בטכנולוגיות שונות של מסדי נתונים ומערכות מחשב. בנוסף לעזרה שסיפק ללקוחות בפתרון צרכים בוערים – תיקון בעיות, ביצוע אופטימיזציה למסדי הנתונים שלהם, הדרכה וייעוץ – איציק סייע לתוכניתנים ולמנהלי מסדי נתונים לעבור לצורת חשיבה רלציונית/מבוססת–סטים, תוך שיפור ביצועי הקוד שלהם וקלות התחזוקה שלהם. תחומי ההתמחות העיקריים של איציק הם שאילתות T-SQL, כוונון שאילתות, תכנות ו–Internals, אך הוא בקי גם בתחומים אחרים של מסד הנתונים. ב–1999 איציק ייסד את קבוצת המשתמשים הישראלית ל– SQL Server ו–OLAP, והוא מנהל אותה עד היום.

לובור קולר (Lubor Kollar)

לובור קולר היה חבר בארגון הפיתוח של SQL Server מאז שחרור הגרסה של SQL Server 6.5 ב–1996. הוא היה מנהל צוות הפיתוח במהלך הפיתוח של SQL Server 2005, והצוות שלו היה אחראי לצד ה"תחתון" של המנוע הרלציוני — מקומפילציה ואופטימיזציה של שאילתות, להרצה של שאילתות, עקביות בטרנזקציות, גיבוי/שחזור וזמינות גבוהה. המאפיינים העיקריים של SQL Server 2005 שעליהם עבד הצוות שלו כוללים חציצה של טבלאות ואינדקסים, Mirroring של מסד הנתונים, Database Snapshot, Snapshot Isolation, שאילתות רקורסיביות ושיפורים אחרים בשאילתות T-SQL,

Database Tuning Advisor ויצירה ותחזוקה Online של אינדקסים. לובור נהנה מהאינטראקציה עם צוות המחקר של מיקרוסופט, שהפכה את תוצאות המחקרים האחרונים למאפיינים במוצר. לפני שהצטרף למיקרוסופט, לובור פיתח מנועי DB2 לפלטפורמות מערכות הפעלה שונות במעבדות IBM בטורונטו קנדה, ובסנטה–תרזה קליפורניה.

בין הישגיו המקצועיים, לובור מעריך במיוחד את העובדה שהצליח לפתור, במהלך שנותיו כסטודנט, את אחת מהבעיות הפתוחות בספר של דונאלד א. קנות' The Art of Computer Programming, Volume 3. וכן, תוכל למצוא את הבעיה והפתרון במהדורה האחרונה של הספר של קנות'.

לובור אוהב את הטבע — סקי, טיולים, גינון, אופני הרים, דייג, ואיסוף פטריות הן הפעילויות האהובות עליו מחוץ לעבודה ולבית. יש לו רישיון כמדריך סקי מקצועי, ובסופי שבוע חורפיים הוא מלמד סקי באתר סקי קרוב לרדמונד, וושינגטון.

מכיוון שלובור אינו שם נפוץ, תוכל להשתמש במנוע החיפוש החביב עליך כדי למצוא פרטים רבים נוספים אודותיו – whitepapers שכתב, תרומות לבלוגים, מאמרים, תקצירים לכנסים, פטנטים, טיולים ועוד.

דיין סרקה (Dejan Sarka)

דיין סרקה — MCP (Microsoft Certified Proffesional), MCDBA (Microsoft Certified DataBase Administrator), MCT (Microsoft Certified Trainre), SQL Server MVP, מנטור ב– Solid Quality Learning — הוא מדריך ויועץ העובד עבור מרכזי הדרכה מוסמכי–מיקרוסופט (CPLS) וחברות פיתוח רבים בסלובניה ובארצות אחרות. בנוסף למתן שירותי הדרכה מקצועיים, הוא עובד באופן קבוע על עיבוד טרנזקציות online (OLTP), OLAP, ופרוייקטים של Data Mining, במיוחד בשלב העיצוב. הוא מרצה קבוע בכמה מהכנסים הבינלאומיים החשובים ביותר, ביניהם TechEd, PASS ו–MCT.

הוא גם תורם חשוב בפגישות Microsoft TechNet אזוריות, בכנס NT שהוא הכנס הגדול ביותר של מיקרוסופט במרכז ומזרח אירופה, ובאירועים נוספים. הוא המייסד של קבוצת המשתמשים הסלובנית של SQL Server. דיין סרקה פיתח גם שני קורסים עבור Solid Quality Learning: האחד הוא Data Modeling Essentials והשני הוא Data Mining with SQL Server 2005.

סטיב קאס (Steve Kass)

סטיב קאס הוא Associate Professor למתמטיקה ומדעי המחשב באוניברסיטת Drew במדיסון, ניו–ג'רזי. סטיב הוא בוגר Pomona College ומחזיק בתואר דוקטור למתמטיקה של אוניברסיטת וויסקונסין–מדיסון. סטיב הוא גם Microsoft SQL Server MVP.

דיויד קמפבל (David Campbell)

דיויד קמפבל הוא המנהל הכללי לאסטרטגיה, תשתית וארכיטקטורה של Microsoft SQL Server.

דיויד סיים תואר שני בהנדסה מכאנית (רובוטיקה) של אוניברסיטת קלארקסון ב–1984 והתחיל לעבוד על תאי–עבודה רובוטיים עבור Sanders Associates (מאוחר יותר אגף של תאגיד Lockheed). ב–1990 הוא הצטרף לתאגיד Digital Equipment, שם עבד על מוצר מסד הנתונים CODASYL שלהם DEC DBMS ועל מוצר מסד הנתונים הרלציוני Rdb.

כאשר הצטרף למיקרוסופט ב–1994, דיויד היה מפתח וארכיטקט בצוות הפיתוח של SQL Server Storage Engine שהיה אחראי בעיקר לכתיבה מחדש של המנוע המרכזי של SQL Server עבור SQL Server גרסה 7.0.

דיויד מחזיק במספר פטנטים בתחומים של ניהול נתונים, סכימה ואיכות תוכנה. הוא מרצה לעיתים קרובות בכנסי התעשייה ובכנסי פיתוח על מגוון נושאים מתחומי ניהול נתונים ופיתוח תוכנה.

אינדקס

לפניך האינדקס ממוין באנגלית. תוכל להשתמש בעברית כמילון וכעזרה להתמצאות בספר.

שים לב, האינדקס מתחיל בעמוד 1 שנמצא בסוף הספר (לפני "על המחברים" והקטלוג המצורף) ומתקדם פנימה אל תוך הספר.

Index

האינדקס ממוין באנגלית. תוכל להשתמש בעברית כמילון וכעזרה להתמצאות בספר.

ת.ד. 6108 הרצליה 46160 **הזמנות באתר**
info@hod-ami.co.il www.**hod-ami**.co.il

קטלוג מרץ 2013 *הנחות ומבצעים באתר*

	כולל	ת. הוצאה	עמ'	מחיר*
אינטרנט - מפתחי אתרים/גרפיקה				
אמא, אבא - בניתי אתר באינטרנט		9/2003	256	69
הגדל את הכנסות העסק שלך באמצעות פרסום בגוגל Google AdWords		6/2010	300	249
מבוא לתכנות בסביבת אינטרנט – הכל כלול בספר אחד HTML + JS + ASP		3/2006	192	69
מבוא לתכנות בסביבת אינטרנט - מבוא ו- **HTML** - חלק 1 מתוך 3 - מהד' 3		6/2003	320	59
מבוא לתכנות בסביבת אינטרנט - **JavaScript** - חלק 2 מתוך 3		5/2002	240	49
מבוא לתכנות בסביבת אינטרנט - **ASP** - חלק 3 מתוך 3 - מהד' 3		3/2006	192	49
HTML5 המדריך לבניית אתרים, הדור הבא (לא בחנויות) חדש		10/2012	352	149
HTML & CSS למפתחי אתרים באינטרנט - מהד' 5		4/2005	592	179
The **Java** Tutorial סדנת לימוד	CD	7/2001	768	179
JavaScript סדנת לימוד		2001	586	159
ASP.NET 3.5 סדנת לימוד בשפות #C ו-VB		10/2009	824	229
תכנות				
לחפש באגים, מדריך מעשי לבודק תוכנה, מהד' 3 (לא בחנויות)		7/2011	350	169
SAS (**Statistical Analysis System**) – ספר לימוד (לא בחנויות)		11/2010	256	169
Visual C# 3.0 סדנת לימוד		9/2008	656	219
ללמוד **C** - מהד' 3		2/2001	480	139
מבוא למדעי המחשב בשפת **C** - מהד' 2 (לא בחנויות)		5/2006	458	119
יסודות התכנות ב-**VBA** לתוכנת **Excel**, מהד' 3 (לא בחנויות)		2012	152	95
PC - חומרה, תוכנה ורשתות				
מדריך Hacking ואבטחת מידע, מהד' 2		9/2011	428	169
מדריך **חומרה ותוכנה** לטכנאי PC - מהד' 5 כולל עדכון 2011	CD	2/2011	752	189
Windows 7/8 נושאים מתקדמים (לא בחנויות. המחיר כולל משלוח)		חדש	140	49
מדריך **רשתות** לטכנאי PC ולמנהלי רשת - מהד' 4		4/2007	608	219
Windows				
Windows 7 צעד-אחר-צעד		1/2010	272	89
Windows 7/8 נושאים מתקדמים (לא בחנויות. המחיר כולל משלוח)		חדש	140	49
Windows XP והכרת המחשב האישי למתחילים		3/2003	208	59
גרפיקה				
Flash – ספר הדרכה ותרגילים (לא בחנויות) חדש		2012	122	64
Illusatrator – ספר הדרכה ותרגילים (לא בחנויות) חדש		2012	120	64
Photoshop צעד אחר צעד (צבע מלא, למתחילים), מהד' 3		3/2012	200	159
Photoshop CS2 עיבוד תמונה דיגיטלית (צבע מלא)		1/2006	480	199
מדריך לתוכנת העיצוב והאנימציה **3ds max** (2 כרכים) (לא בחנויות)	CD	מהד' 5	1400	419

* מחיר מומלץ לצרכן כולל מע"מ. **יש להיכנס לאתר כדי לבדוק מבצעים ומחירים מיוחדים**

היכנס לאתר להתעדכן בספרים החדשים ומבצעים www.**hod-ami**.co.il

		כולל	הוצאה	עמ'	מחיר*
OFFICE 2010					
טבלאות ציר – ניתוח נתונים חכם	(לא בחנויות)		2012	116	87
יסודות התכנות ב-**VBA** לתוכנת **Excel**, מהד' 3	(לא בחנויות)		2012	152	95
Word 2010 צעד אחר צעד			11/2010	160	69
Excel 2010 סדנת לימוד			11/2010	344	129
PowerPoint 2010 צעד אחר צעד			11/2010	202	69
Outlook 2010 צעד אחר צעד			11/2010	190	69
Access 2010 סדנת לימוד			11/2010	360	179
OFFICE 2007					
PowerPoint 2007 למתחילים			8/2007	240	99
Outlook 2007 למתחילים			1/2008	224	99
OFFICE 2003					
Word 2003 צעד אחר צעד			3/2004	208	69
Excel 2003 הסדרה הידידותית למתחילים			2/2004	160	49
PowerPoint 2003 הסדרה הידידותית למתחילים			2/2004	96	49
Outlook 2003 הסדרה הידידותית למתחילים			2/2004	96	49
Access 2003 הסדרה הידידותית למתחילים			2/2004	144	19
OFFICE XP					
Office XP צעד אחר צעד		CD	7/2001	576	139
Word XP הסדרה הידידותית למתחילים			7/2001	144	49
ניהול, כלכלה ושונות					
לחפש באגים, מדריך מעשי לבודק תוכנה, מהד' 3	(לא בחנויות)		7/2011	350	169
חקר ביצועים כשירות ללקוח			2/2010	236	92
ניהול ממוקד לעשות יותר עם מה שיש (כריכה קשה) - מהד' 4	חדש		9/2012	358	133
לי זה עולה יותר (תמחיר) (כריכה קשה) - מהד' 3			10/2006	368	129
מערכות מידע					
Oracle SQL יכולות מתקדמות	(לא בחנויות)		7/2011	626	249
SAS (**Statistical Analysis System**) – ספר לימוד	(לא בחנויות)		11/2010	256	169
בסיסי נתונים ושפת **SQL** – עקרונות ועיצוב	(לא בחנויות)			648	149
ניתוח מערכות מידע כולל מתודולוגיית ה- **UML**	(לא בחנויות)		11/2004	818	229
המדריך העברי השלם **UML**	(לא בחנויות)		1/2009	346	329

* מחיר מומלץ לצרכן כולל מע"מ. קטלוג 3/2013.

<u>יש להיכנס לאתר כדי לבדוק מבצעים ומחירים מיוחדים</u>

היכנס לאתר להתעדכן בספרים החדשים ומבצעים

תוכן עניינים ופרקים לדוגמה www.**hod-ami**.co.il