T-SQL Querying

Itzik Ben-Gan
Dejan Sarka
Adam Machanic
Kevin Farlee
To Lilach, for giving meaning to everything that I do.

—Itzik
Contents at a glance

Foreword xv
Introduction xvii

CHAPTER 1 Logical query processing 1
CHAPTER 2 Query tuning 41
CHAPTER 3 Multi-table queries 187
CHAPTER 4 Grouping, pivoting, and windowing 259
CHAPTER 5 TOP and OFFSET-FETCH 341
CHAPTER 6 Data modification 373
CHAPTER 7 Working with date and time 419
CHAPTER 8 T-SQL for BI practitioners 473
CHAPTER 9 Programmable objects 525
CHAPTER 10 In-Memory OLTP 671
CHAPTER 11 Graphs and recursive queries 707

Index 803
# Contents

*Foreword* .......................................................... xv

*Introduction* .................................................... xvii

## Chapter 1  Logical query processing  

Logical query-processing phases ........................................ 3
  Logical query-processing phases in brief ............................. 4

Sample query based on customers/orders scenario ..................... 6

Logical query-processing phase details ................................. 8
  Step 1: The FROM phase ........................................ 8
  Step 2: The WHERE phase ...................................... 14
  Step 3: The GROUP BY phase .................................. 15
  Step 4: The HAVING phase ..................................... 16
  Step 5: The SELECT phase ..................................... 17
  Step 6: The ORDER BY phase ................................... 20
  Step 7: Apply the TOP or OFFSET-FETCH filter .................. 22

Further aspects of logical query processing ............................. 26
  Table operators ................................................. 26
  Window functions ............................................... 35
  The UNION, EXCEPT, and INTERSECT operators ................. 38

Conclusion ...................................................................... 39

## Chapter 2  Query tuning  

Internals ................................................................. 41
  Pages and extents ................................................. 42
  Table organization ................................................ 43

Tools to measure query performance ..................................... 53

---

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: 

[microsoft.com/learning/booksurvey](microsoft.com/learning/booksurvey)
## Access methods
- Table scan/unordered clustered index scan ........................................ 57
- Unordered covering nonclustered index scan ...................................... 60
- Ordered clustered index scan ............................................................ 62
- Ordered covering nonclustered index scan ......................................... 63
- The storage engine’s treatment of scans ............................................. 65
- Nonclustered index seek + range scan + lookups .................................. 81
- Unordered nonclustered index scan + lookups ..................................... 91
- Clustered index seek + range scan ...................................................... 93
- Covering nonclustered index seek + range scan ................................... 94

## Cardinality estimates
- Legacy estimator vs. 2014 cardinality estimator .................................. 98
- Implications of underestimations and overestimations ......................... 99
- Statistics ......................................................................................... 101
- Estimates for multiple predicates ...................................................... 104
- Ascending key problem .................................................................... 107
- Unknowns ....................................................................................... 110

## Indexing features
- Descending indexes ................................................................. 115
- Included non-key columns ............................................................ 119
- Filtered indexes and statistics .......................................................... 120
- Columnstore indexes ................................................................. 123
- Inline index definition ................................................................. 130

## Prioritizing queries for tuning with extended events
- Index and query information and statistics ....................................... 134
- Temporary objects ........................................................................ 139
- Set-based vs. iterative solutions ...................................................... 149
- Query tuning with query revisions .................................................. 153
- Parallel query execution ............................................................... 158
  - How intraquery parallelism works .................................................. 158
  - Parallelism and query optimization .............................................. 175
  - The parallel APPLY query pattern .............................................. 181

## Conclusion ................................................................. 186
**Chapter 3**  
**Multi-table queries**  

<table>
<thead>
<tr>
<th>Subqueries</th>
<th>187</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-contained subqueries</td>
<td>187</td>
</tr>
<tr>
<td>Correlated subqueries</td>
<td>189</td>
</tr>
<tr>
<td>The EXISTS predicate</td>
<td>194</td>
</tr>
<tr>
<td>Misbehaving subqueries</td>
<td>201</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table expressions</th>
<th>204</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived tables</td>
<td>205</td>
</tr>
<tr>
<td>CTEs</td>
<td>207</td>
</tr>
<tr>
<td>Views</td>
<td>211</td>
</tr>
<tr>
<td>Inline table-valued functions</td>
<td>215</td>
</tr>
<tr>
<td>Generating numbers</td>
<td>215</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The APPLY operator</th>
<th>218</th>
</tr>
</thead>
<tbody>
<tr>
<td>The CROSS APPLY operator</td>
<td>219</td>
</tr>
<tr>
<td>The OUTER APPLY operator</td>
<td>221</td>
</tr>
<tr>
<td>Implicit APPLY</td>
<td>221</td>
</tr>
<tr>
<td>Reuse of column aliases</td>
<td>222</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Joins</th>
<th>224</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross join</td>
<td>224</td>
</tr>
<tr>
<td>Inner join</td>
<td>228</td>
</tr>
<tr>
<td>Outer join</td>
<td>229</td>
</tr>
<tr>
<td>Self join</td>
<td>230</td>
</tr>
<tr>
<td>Equi and non-equijoin</td>
<td>230</td>
</tr>
<tr>
<td>Multi-join queries</td>
<td>231</td>
</tr>
<tr>
<td>Semi and anti semi join</td>
<td>237</td>
</tr>
<tr>
<td>Join algorithms</td>
<td>239</td>
</tr>
<tr>
<td>Separating elements</td>
<td>245</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The UNION, EXCEPT, and INTERSECT operators</th>
<th>249</th>
</tr>
</thead>
<tbody>
<tr>
<td>The UNION ALL and UNION operators</td>
<td>250</td>
</tr>
<tr>
<td>The INTERSECT operator</td>
<td>253</td>
</tr>
<tr>
<td>The EXCEPT operator</td>
<td>255</td>
</tr>
</tbody>
</table>

| Conclusion | 257 |
Chapter 4  Grouping, pivoting, and windowing  259

Window functions .......................................................... 259
  Aggregate window functions ........................................... 260
  Ranking window functions ............................................. 281
  Offset window functions ............................................... 285
  Statistical window functions .......................................... 288
  Gaps and islands .............................................................. 291

Pivoting ................................................................. 299
  One-to-one pivot .......................................................... 300
  Many-to-one pivot ........................................................ 304

Unpivoting ............................................................... 307
  Unpivoting with CROSS JOIN and VALUES ....................... 308
  Unpivoting with CROSS APPLY and VALUES ...................... 310
  Using the UNPIVOT operator ........................................... 312

Custom aggregations ..................................................... 313
  Using a cursor ............................................................ 314
  Using pivoting ........................................................... 315
  Specialized solutions .................................................... 316

Grouping sets ............................................................. 327
  GROUPING SETS subclause ............................................. 328
  CUBE and ROLLUP clauses .............................................. 331
  Grouping sets algebra .................................................... 333
  Materializing grouping sets .......................................... 334
  Sorting ................................................................. 337

Conclusion ................................................................. 339

Chapter 5  TOP and OFFSET-FETCH  341

The TOP and OFFSET-FETCH filters ............................... 341
  The TOP filter ........................................................... 341
  The OFFSET-FETCH filter .............................................. 345

Optimization of filters demonstrated through paging 346
  Optimization of TOP ..................................................... 346
Optimization of OFFSET-FETCH ..................................................354
Optimization of ROW_NUMBER ............................................358
Using the TOP option with modifications .................................360
TOP with modifications ...........................................................360
Modifying in chunks ...............................................................361
Top N per group .....................................................................363
Solution using ROW_NUMBER .................................................364
Solution using TOP and APPLY .................................................365
Solution using concatenation (a carry-along sort) .......................366
Median .................................................................................368
Solution using PERCENTILE_CONT .........................................369
Solution using ROW_NUMBER .................................................369
Solution using OFFSET-FETCH and APPLY ...............................370
Conclusion .............................................................................371

Chapter 6  Data modification .................................................373
Inserting data ........................................................................373
SELECT INTO .........................................................................373
Bulk import .............................................................................376
Measuring the amount of logging .............................................377
BULK rowset provider .............................................................378
Sequences .............................................................................381
Characteristics and inflexibilities of the identity property ............381
The sequence object ...............................................................382
Performance considerations ......................................................387
Summarizing the comparison of identity with sequence .............394
Deleting data ........................................................................395
TRUNCATE TABLE .................................................................395
Deleting duplicates .................................................................399
Updating data ........................................................................401
Update using table expressions ...............................................402
Update using variables ............................................................403
Merging data ................................................................. 404
MERGE examples ...................................................... 405
Preventing MERGE conflicts ........................................ 408
ON isn’t a filter .......................................................... 409
USING is similar to FROM ....................................... 410
The OUTPUT clause .................................................. 411
Example with INSERT and identity ............................. 412
Example for archiving deleted data ............................. 413
Example with the MERGE statement ......................... 414
Composable DML ..................................................... 417
Conclusion .................................................................... 417

Chapter 7 Working with date and time 419
Date and time data types ................................................. 419
Date and time functions ............................................... 422
Challenges working with date and time ......................... 434
  Literals ..................................................................... 434
  Identifying weekdays ................................................ 436
  Handling date-only or time-only data with DATETIME and
  SMALLDATETIME .................................................. 439
  First, last, previous, and next date calculations ............ 440
  Search argument ..................................................... 445
  Rounding issues ..................................................... 447
Querying date and time data ........................................... 449
  Grouping by the week ............................................. 449
  Intervals .................................................................. 450
Conclusion .................................................................... 471

Chapter 8 T-SQL for BI practitioners 473
Data preparation .......................................................... 473
Sales analysis view ...................................................... 474
Frequencies .................................................................. 476
  Frequencies without window functions .................... 476
Frequencies with window functions .............................. 477
Descriptive statistics for continuous variables .................... 479
Centers of a distribution ........................................... 479
Spread of a distribution .......................................... 482
Higher population moments ...................................... 487
Linear dependencies .................................................. 495
Two continuous variables ......................................... 495
Contingency tables and chi-squared ............................... 501
Analysis of variance ............................................... 505
Definite integration .................................................. 509
Moving averages and entropy ...................................... 512
Moving averages ...................................................... 512
Entropy ..................................................................... 518
Conclusion ................................................................... 522

Chapter 9  Programmable objects .................................. 525
Dynamic SQL .................................................................. 525
Using the EXEC command ............................................. 525
Using the sp_executesql procedure ................................ 529
Dynamic pivot .............................................................. 530
Dynamic search conditions ......................................... 535
Dynamic sorting .......................................................... 542
User-defined functions ................................................... 546
Scalar UDFs ................................................................. 546
Multistatement TVFs ..................................................... 550
Stored procedures ........................................................ 553
Compilations, recompilations, and reuse of execution plans .... 554
Table type and table-valued parameters ........................... 571
EXECUTE WITH RESULT SETS .................................... 573
Triggers ...................................................................... 575
Trigger types and uses .................................................. 575
Efficient trigger programming ....................................... 581
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCLR programming</td>
<td>585</td>
</tr>
<tr>
<td>SQLCLR architecture</td>
<td>586</td>
</tr>
<tr>
<td>CLR scalar functions and creating your first assembly</td>
<td>588</td>
</tr>
<tr>
<td>Streaming table-valued functions</td>
<td>597</td>
</tr>
<tr>
<td>SQLCLR stored procedures and triggers</td>
<td>605</td>
</tr>
<tr>
<td>SQLCLR user-defined types</td>
<td>617</td>
</tr>
<tr>
<td>SQLCLR user-defined aggregates</td>
<td>628</td>
</tr>
<tr>
<td>Transaction and concurrency</td>
<td>632</td>
</tr>
<tr>
<td>Transactions described</td>
<td>633</td>
</tr>
<tr>
<td>Locks and blocking</td>
<td>636</td>
</tr>
<tr>
<td>Lock escalation</td>
<td>641</td>
</tr>
<tr>
<td>Delayed durability</td>
<td>643</td>
</tr>
<tr>
<td>Isolation levels</td>
<td>645</td>
</tr>
<tr>
<td>Deadlocks</td>
<td>657</td>
</tr>
<tr>
<td>Error handling</td>
<td>662</td>
</tr>
<tr>
<td>The TRY-CATCH construct</td>
<td>662</td>
</tr>
<tr>
<td>Errors in transactions</td>
<td>666</td>
</tr>
<tr>
<td>Retry logic</td>
<td>669</td>
</tr>
<tr>
<td>Conclusion</td>
<td>670</td>
</tr>
</tbody>
</table>

**Chapter 10 In-Memory OLTP** 671

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-Memory OLTP overview</td>
<td>671</td>
</tr>
<tr>
<td>Data is always in memory</td>
<td>672</td>
</tr>
<tr>
<td>Native compilation</td>
<td>673</td>
</tr>
<tr>
<td>Lock and latch-free architecture</td>
<td>673</td>
</tr>
<tr>
<td>SQL Server integration</td>
<td>674</td>
</tr>
<tr>
<td>Creating memory-optimized tables</td>
<td>675</td>
</tr>
<tr>
<td>Creating indexes in memory-optimized tables</td>
<td>676</td>
</tr>
<tr>
<td>Clustered vs. nonclustered indexes</td>
<td>677</td>
</tr>
<tr>
<td>Nonclustered indexes</td>
<td>677</td>
</tr>
<tr>
<td>Hash indexes</td>
<td>680</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Execution environments</td>
<td>690</td>
</tr>
<tr>
<td>Query interop</td>
<td>690</td>
</tr>
<tr>
<td>Natively compiled procedures</td>
<td>699</td>
</tr>
<tr>
<td>Surface-area restrictions</td>
<td>703</td>
</tr>
<tr>
<td>Table DDL</td>
<td>703</td>
</tr>
<tr>
<td>DML</td>
<td>704</td>
</tr>
<tr>
<td>Conclusion</td>
<td>705</td>
</tr>
<tr>
<td>Chapter 11 Graphs and recursive queries</td>
<td>707</td>
</tr>
<tr>
<td>Terminology</td>
<td>707</td>
</tr>
<tr>
<td>Graphs</td>
<td>707</td>
</tr>
<tr>
<td>Trees</td>
<td>708</td>
</tr>
<tr>
<td>Hierarchies</td>
<td>709</td>
</tr>
<tr>
<td>Scenarios</td>
<td>709</td>
</tr>
<tr>
<td>Employee organizational chart</td>
<td>709</td>
</tr>
<tr>
<td>Bill of materials (BOM)</td>
<td>711</td>
</tr>
<tr>
<td>Road system</td>
<td>715</td>
</tr>
<tr>
<td>Iteration/recursion</td>
<td>718</td>
</tr>
<tr>
<td>Subgraph/descendants</td>
<td>719</td>
</tr>
<tr>
<td>Ancestors/path</td>
<td>730</td>
</tr>
<tr>
<td>Subgraph/descendants with path enumeration</td>
<td>733</td>
</tr>
<tr>
<td>Sorting</td>
<td>736</td>
</tr>
<tr>
<td>Cycles</td>
<td>740</td>
</tr>
<tr>
<td>Materialized path</td>
<td>742</td>
</tr>
<tr>
<td>Maintaining data</td>
<td>743</td>
</tr>
<tr>
<td>Querying</td>
<td>749</td>
</tr>
<tr>
<td>Materialized path with the HIERARCHYID data type</td>
<td>754</td>
</tr>
<tr>
<td>Maintaining data</td>
<td>756</td>
</tr>
<tr>
<td>Querying</td>
<td>763</td>
</tr>
<tr>
<td>Further aspects of working with HIERARCHYID</td>
<td>767</td>
</tr>
<tr>
<td>Nested sets</td>
<td>778</td>
</tr>
<tr>
<td>Assigning left and right values</td>
<td>778</td>
</tr>
<tr>
<td>Querying</td>
<td>784</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>787</td>
</tr>
<tr>
<td>Directed acyclic graph</td>
<td>787</td>
</tr>
<tr>
<td>Conclusion</td>
<td>801</td>
</tr>
<tr>
<td>Index</td>
<td>803</td>
</tr>
</tbody>
</table>
Foreword

I have been with Microsoft and working with the Microsoft SQL Server team since 1993. It has been one heck of a ride to watch this product mature into what it is today. It has been exciting to watch how the Microsoft SQL Server customer base uses SQL Server to run their mission-critical businesses. Most of all, it has been an honor to support the most vibrant, passionate technology community I have ever seen.

The Microsoft SQL Server community is filled with truly amazing, smart people. They take pride in sharing their great knowledge with others, all for making the community stronger. Anyone in the world can jump on Twitter and ask any question to #sqlhelp, and within seconds one of the smartest experts in the world will be responding. If you are looking for expertise in performance, storage, query optimization, large-scale design, modeling, or any data-related topic, these experts are in the community today sharing their expertise. You will get to know them not only by their expertise but by their unique, friendly personalities as well. We in the SQL Server community world refer to this as our SQL family.

Everyone in the community knows the major contributors by their expertise in particular areas. If you ask who the best database performance expert is, people in the community will give you the same four or five names. If you ask for the best storage expert, again people will give you the same four or five storage expert names. You’ll always find a few experts in the community who are the very best for a specific area of database domain expertise. There is only one exception to this that I am aware of, and that is the T-SQL language. There are a lot of talented T-SQL experts, but if you ask for the best everyone will give you one name: Itzik Ben-Gan.

Itzik asked me to write this foreword for his new book, and I am honored to do so. His previous books—Inside Microsoft SQL Server: T-SQL Querying (Microsoft Press, 2009), Inside Microsoft SQL Server: T-SQL Programming (Microsoft Press, 2009), and Microsoft SQL Server High-Performance T-SQL Using Window Functions (Microsoft Press, 2012)—are sitting on the shelves of every DBA I know. These books add up to over 2,000 pages of top-notch technical knowledge about Microsoft SQL Server T-SQL, and they set the standard for high-quality database content.

I am excited about this new book, T-SQL Querying. Not only does it combine material from his three previous books, but it also adds material from SQL Server 2012 and 2014, including window functions, the new cardinality estimator, sequences, column-store, In-Memory OLTP, and much more. Itzik has a few exciting co-authors as well: Kevin Farlee, Adam Machanic, and Dejan Sarka. Kevin is part of the Microsoft SQL
Server engineering team and someone I have been working with for many years. Adam is one of those few names that I refer to above as one of the best database performance experts in the world, and Dejan is well known for his BI and data-modeling expertise.

I fully expect this book to be the standard T-SQL guide for the Microsoft SQL Server community.

*Mark Souza*
*General Manager, Cloud and Enterprise Engineering*
*Microsoft*
Introduction

Updating both *Inside Microsoft SQL Server 2008: T-SQL Querying* (Microsoft Press, 2009) and parts of *Inside Microsoft SQL Server 2008: T-SQL Programming* (Microsoft Press, 2009), this book gives database developers and administrators a detailed look at the internal architecture of T-SQL and a comprehensive programming reference. It includes coverage of SQL Server 2012 and 2014, but in many cases deals with areas that are not version-specific and will likely be relevant in future versions of SQL Server. Tackle the toughest set-based querying and query-tuning problems—guided by an author team with in-depth, inside knowledge of T-SQL. Deepen your understanding of architecture and internals—and learn practical approaches and advanced techniques to optimize your code’s performance. This book covers many unique techniques that were developed, improved, and polished by the authors over their many years of experience, providing highly efficient solutions for common challenges. There’s a deep focus on the performance and efficiency of the techniques and solutions covered. The book also emphasizes the need to have a correct understanding of the language and its underlying mathematical foundations.

Who should read this book

This book is designed to help experienced T-SQL practitioners become more knowledgeable and efficient in this field. The book’s target audience is T-SQL developers, DBAs, BI pros, data scientists, and anyone who is serious about T-SQL. Its main purpose is to prepare you for real-life needs, as far as T-SQL is concerned. Its main focus is not to help you pass certification exams. That said, it just so happens that the book covers many of the topics that exams 70-461 and 70-464 test you on. So, even though you shouldn’t consider this book as the only learning tool to prepare for these exams, it is certainly a tool that will help you in this process.

Assumptions

This book assumes that you have at least a year of solid experience working with SQL Server, writing and tuning T-SQL code. It assumes that you have a good grasp of T-SQL coding and tuning fundamentals, and that you are ready to tackle more advanced challenges. This book could still be relevant to you if you have similar experience with a different database platform and its dialect of SQL, but actual knowledge and experience with SQL Server and T-SQL is preferred.
This book might not be for you if...

This book might not be for you if you are fairly new to databases and SQL.

Organization of this book

The book starts with two chapters that lay the foundation of logical and physical query processing required to gain the most from the rest of the chapters.

The first chapter covers logical query processing. It describes in detail the logical phases involved in processing queries, the unique aspects of SQL querying, and the special mindset you need to adopt to program in a relational, set-oriented environment.

The second chapter covers query tuning and the physical layer. It describes internal data structures, tools to measure query performance, access methods, cardinality estimates, indexing features, prioritizing queries with extended events, columnstore technology, use of temporary tables and table variables, sets versus cursors, query tuning with query revisions, and parallel query execution. (The part about parallel query execution was written by Adam Machanic.)

The next five chapters deal with various data manipulation–related topics. The coverage of these topics is extensive; beyond explaining the features, they focus a lot on the performance of the code and the use of the features to solve common tasks. Chapter 3 covers multi-table queries using subqueries, the APPLY operator, joins, and the UNION, INTERSECT, and EXCEPT relational operators. Chapter 4 covers data analysis using grouping, pivoting, and window functions. Chapter 5 covers the TOP and OFFSET-FETCH filters, and solving top N per group tasks. Chapter 6 covers data-modification topics like minimally logged operations, using the sequence object efficiently, merging data, and the OUTPUT clause. Chapter 7 covers date and time treatment, including the handling of date and time intervals.

Chapter 8 covers T-SQL for BI practitioners and was written by Dejan Sarka. It describes how to prepare data for analysis and how to use T-SQL to handle statistical data analysis tasks. Those include frequencies, descriptive statistics for continuous variables, linear dependencies, and moving averages and entropy.

Chapter 9 covers the programmability constructs that T-SQL supports. Those are dynamic SQL, user-defined functions, stored procedures, triggers, SQLCLR programming (written by Adam Machanic), transactions and concurrency, and error handling.
Previously, these topics were covered in the book *Inside Microsoft SQL Server: T-SQL Programming*.

Chapter 10 covers one of the major improvements in SQL Server 2014—the In-Memory OLTP engine. This chapter was written by Microsoft’s Kevin Farlee, who was involved in the actual development of this feature.

Chapter 11 covers graphs and recursive queries. It shows how to handle graph structures such as employee hierarchies, bill of materials, and maps in SQL Server using T-SQL. It shows how to implement models such as the enumerated path model (using your own custom solution and using the HIERARCHYID data type) and the nested sets model. It also shows how to use recursive queries to manipulate data in graphs.

**System requirements**

You will need the following software to run the code samples in this book:

- Microsoft SQL Server 2014:
  - Edition: 64-bit Enterprise, Developer, or Evaluation; other editions do not support the In-Memory OLTP and columnstore technologies that are covered in the book. You can download a trial version here: [http://www.microsoft.com/sql](http://www.microsoft.com/sql).
  - In the Feature Selection dialog of the SQL Server 2014 setup program, choose the following components: Database Engine Services, Client Tools Connectivity, Documentation Components, Management Tools – Basic, Management Tools – Complete.

- Microsoft Visual Studio 2013 with Microsoft SQL Server Data Tools (SSDT):

Depending on your Windows configuration, you might require Local Administrator rights to install or configure SQL Server 2014 and Visual Studio 2013.
Downloads: Code samples

This book contains many code samples. You can download the source code for this book from the authors’ site: http://tsql.solidq.com/books/tq3.

The source code is organized in a compressed file named T-SQL Querying - YYYYMMDD.zip, where YYYYMMDD stands for the last update date of the content. Follow the instructions in the Readme.txt file that is included in the compressed file to install the code samples.

Acknowledgments

A number of people contributed to making this book a reality, whether directly or indirectly, and deserve thanks and recognition. It’s certainly possible that I omitted some names unintentionally and I apologize for this ahead of time.

To Lilach: you’re the one who makes me want to be good at what I do. Besides being my inspiration in life, you had an active unofficial role in this book. You were the book’s first reader! So many hours we spent reading the text together looking for errors before I sent it to the editors. I have a feeling that you know some things about T-SQL better than people who are professionals in the field.

To my parents, Mila and Gabi, and to my siblings, Mickey and Ina, for the constant support and for accepting the fact that I’m away. You experienced so much turbulence in the last few years, and I’m hoping that the coming years will be healthy and happy.

To the coauthors of the book, Dejan Sarka, Adam Machanic, and Kevin Farlee. It’s a true privilege to be part of such a knowledgeable and experienced group of people. Each of you are such experts in your areas that I felt that your topics would be best served if covered by you: Dejan with the chapter on T-SQL for BI practitioners, Adam with the sections on parallel query execution and SQL CLR programming, and Kevin with the chapter on In-Memory OLTP. Thanks for taking part in this book.

To the technical reviewer of the book, Alejandro Mesa: you read and unofficially reviewed my previous books. You’re so passionate about the topic that I’m glad with this book you took a more official reviewer role. I also want to thank the reviewer of the former edition of the book, Steve Kass: you did such thorough and brilliant work that a lot of it echoes in this one.
To Mark Souza: you were there pretty much since the inception of the product, being involved in technical, management, and community-related roles. If anyone feels the heartbeat of the SQL Server community, it is you. We’re all grateful for what you do, and it is a great honor to have you write the foreword.

Many thanks to the editors at Microsoft Press. To Devon Musgrave, who played both the acquisitions editor and developmental editor roles: you are the one who made this book a reality and handled all the initial stages. I realize that this book is very likely one of many you were responsible for, and I’d like to thank you for dedicating the time and effort that you did. To Carol Dillingham, the book’s project editor: you spent so many hours on this project, and you coordinated it delightfully. It was a pleasure working with you. Also, thanks to Curtis Philips, the project manager from Publishing.com. It was a complex project, and I’m sure it wasn’t a picnic for you. Also, many thanks to the copyeditor, Roger LeBlanc, who worked on my previous books, and to the proofreader, Andrea Fox. It was a pleasure to work with you guys.

To SolidQ, my company for over a decade: it’s gratifying to be part of such a great company that evolved into what it is today. The members of this company are much more than colleagues to me; they are partners, friends, and family. Thanks to Fernando G. Guerrero, Douglas McDowell, Herbert Albert, Dejan Sarka, Gianluca Hotz, Antonio Soto, Jeanne Reeves, Glenn McCoin, Fritz Lechnitz, Eric Van Soldt, Berry Walker, Marilyn Templeton, Joelle Budd, Gwen White, Jan Taylor, Judy Dyess, Alberto Martin, Lorena Jimenez, Ron Talmage, Andy Kelly, Rushabh Mehta, Joe Chang, Mark Tabladillo, Eladio Rincón, Miguel Egea, Alejandro J. Rocchi, Daniel A. Seara, Javier Loria, Paco González, Enrique Catalá, Esther Nolasco Andreu, Rocío Guerrero, Javier Torrenteras, Rubén Garrigós, Victor Vale Diaz, Davide Mauri, Danilo Dominici, Erik Veerman, Jay Hackney, Grega Jerkič, Matija Lah, Richard Waymire, Carl Rabeler, Chris Randall, Tony Rogerson, Christian Rise, Raoul Illyés, Johan Åhlén, Peter Larsson, Paul Turley, Bill Haenlin, Blythe Gietz, Nigel Semmi, Paras Doshi, and so many others.

To members of the Microsoft SQL Server development team, past and present: Tobias Ternstrom, Lubor Kollar, Umachandar Jayachandran (UC), Boris Baryshnikov, Conor Cunningham, Kevin Farlee, Marc Friedman, Milan Stojic, Craig Freedman, Campbell Fraser, Mark Souza, T. K. Rengarajan, Dave Campbell, César Galindo-Legaria, and I’m sure many others. I know it wasn’t a trivial effort to add support for window functions in SQL Server. Thanks for the great effort, and thanks for all the time you spent meeting with me and responding to my emails, addressing my questions, and answering my requests for clarification.
To members of the SQL Server Pro editorial team, past and present: Megan Keller, Lavon Peters, Michele Crockett, Mike Otey, Jayleen Heft, and I’m sure many others. I’ve been writing for the magazine for over a decade and a half, and I am grateful for the opportunity to share my knowledge with the magazine’s readers.

To SQL Server MVPs, past and present: Paul White, Alejandro Mesa, Erland Sommarskog, Aaron Bertrand, Tibor Karaszi, Benjamin Nevarez, Simon Sabin, Darren Green, Allan Mitchell, Tony Rogerson, and many others—and to the MVP lead, Simon Tien. This is a great program that I’m grateful and proud to be part of. The level of expertise of this group is amazing, and I’m always excited when we all get to meet, both to share ideas and just to catch up at a personal level over beer. I have to extend special thanks to Paul White. I’ve learned so much from you, and I thoroughly enjoy reading your work. I think it’s safe to say that you’re my favorite author. Who knows, maybe one day we’ll get to work on something together.

Finally, to my students: teaching about T-SQL is what drives me. It’s my passion. Thanks for allowing me to fulfill my calling and for all the great questions that make me seek more knowledge.

—Cheers, Itzik

Errata, updates, & book support

We’ve made every effort to ensure the accuracy of this book. If you discover an error, please submit it to us via mspinput@microsoft.com. You can also reach the Microsoft Press Book Support team for other assistance via the same email address. Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to http://support.microsoft.com.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!
We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you’re busy, so we’ve kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let’s keep the conversation going! We’re on Twitter: http://twitter.com/MicrosoftPress.
This page intentionally left blank
CLASSIC FILTERS IN SQL LIKE ON, WHERE, AND HAVING ARE BASED ON PREDICATES. TOP AND OFFSET-FETCH ARE FILTERS THAT ARE BASED ON A DIFFERENT CONCEPT: YOU INDICATE ORDER AND HOW MANY ROWS TO FILTER BASED ON THAT ORDER. MANY FILTERING TASKS ARE DEFINED BASED ON ORDER AND A REQUIRED NUMBER OF ROWS. IT’S CERTAINLY GOOD TO HAVE LANGUAGE SUPPORT IN T-SQL THAT ALLOWS YOU TO PHRASE THE REQUEST IN A MANNER THAT IS SIMILAR TO THE WAY YOU THINK ABOUT THE TASK.

THIS CHAPTER STARTS WITH THE LOGICAL DESIGN ASPECTS OF THE FILTERS. IT THEN USES A PAGING SCENARIO TO DEMONSTRATE THEIR OPTIMIZATION. THE CHAPTER ALSO COVERS THE USE OF TOP WITH MODIFICATION STATEMENTS. FINALLY, THE CHAPTER DEMONSTRATES THE USE OF TOP AND OFFSET-FETCH IN SOLVING PRACTICAL PROBLEMS LIKE TOP N PER GROUP AND MEDIAN.

THE TOP AND OFFSET-FETCH FILTERS

YOU USE THE TOP AND OFFSET-FETCH FILTERS TO IMPLEMENT FILTERING REQUIREMENTS IN YOUR QUERIES IN AN INTUITIVE MANNER. THE TOP FILTER IS A PROPRIETARY FEATURE IN T-SQL, WHEREAS THE OFFSET-FETCH FILTER IS A STANDARD FEATURE. T-SQL STARTED SUPPORTING OFFSET-FETCH WITH MICROSOFT SQL SERVER 2012. AS OF SQL SERVER 2014, THE IMPLEMENTATION OF OFFSET-FETCH IN T-SQL IS STILL MISSING A COUPLE OF STANDARD ELEMENTS—INTERESTINGLY, ONES THAT ARE AVAILABLE WITH TOP. WITH THE CURRENT IMPLEMENTATION, EACH OF THE FILTERS HAS CAPABILITIES THAT ARE NOT SUPPORTED BY THE OTHER.

I’LL START BY DESCRIBING THE LOGICAL DESIGN ASPECTS OF TOP AND THEN COVER THOSE OF OFFSET-FETCH.

THE TOP FILTER

THE TOP FILTER IS A COMMONLY USED CONSTRUCT IN T-SQL. ITS POPULARITY PROBABLY CAN BE ATTRIBUTED TO THE FACT THAT ITS DESIGN IS SO WELL ALIGNED WITH THE WAY MANY FILTERING REQUIREMENTS ARE EXPRESSED—FOR EXAMPLE, “RETURN THE THREE MOST RECENT ORDERS.” IN THIS REQUEST, THE ORDER FOR THE FILTER IS BASED ON ORDERDATE, DESCENDING, AND THE NUMBER OF ROWS YOU WANT TO FILTER BASED ON THIS ORDER IS 3.

YOU SPECIFY THE TOP OPTION IN THE SELECT LIST WITH AN INPUT VALUE TYPED AS BIGINT INDICATING HOW MANY ROWS YOU WANT TO FILTER. YOU PROVIDE THE ORDERING SPECIFICATION IN THE CLASSIC ORDER BY CLAUSE. FOR EXAMPLE, YOU USE THE FOLLOWING QUERY TO GET THE THREE MOST RECENT ORDERS.
USE TSQLV3;

SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;

I got the following output from this query:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>11077</td>
<td>2015-05-06</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>11076</td>
<td>2015-05-06</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>11075</td>
<td>2015-05-06</td>
<td>68</td>
<td>8</td>
</tr>
</tbody>
</table>

Instead of specifying the number of rows you want to filter, you can use TOP to specify the percent (of the total number of rows in the query result). You do so by providing a value in the range 0 through 100 (typed as FLOAT) and add the PERCENT keyword. For example, in the following query you request to filter one percent of the rows:

SELECT TOP (1) PERCENT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;

SQL Server rounds up the number of rows computed based on the input percent. For example, the result of 1 percent applied to 830 rows in the Orders table is 8.3. Rounding up this number, you get 9. Here’s the output I got for this query:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>11074</td>
<td>2015-05-06</td>
<td>73</td>
<td>7</td>
</tr>
<tr>
<td>11075</td>
<td>2015-05-06</td>
<td>68</td>
<td>8</td>
</tr>
<tr>
<td>11076</td>
<td>2015-05-06</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>11077</td>
<td>2015-05-06</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>11070</td>
<td>2015-05-05</td>
<td>44</td>
<td>2</td>
</tr>
<tr>
<td>11071</td>
<td>2015-05-05</td>
<td>46</td>
<td>1</td>
</tr>
<tr>
<td>11072</td>
<td>2015-05-05</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>11073</td>
<td>2015-05-05</td>
<td>58</td>
<td>2</td>
</tr>
<tr>
<td>11067</td>
<td>2015-05-04</td>
<td>17</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that to translate the input percent to a number of rows, SQL Server has to first figure out the count of rows in the query result, and this usually requires extra work.

Interestingly, ordering specification is optional for the TOP filter. For example, consider the following query:

SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders;
I got the following output from this query:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>10248</td>
<td>2013-07-04</td>
<td>85</td>
<td>5</td>
</tr>
<tr>
<td>10249</td>
<td>2013-07-05</td>
<td>79</td>
<td>6</td>
</tr>
<tr>
<td>10250</td>
<td>2013-07-08</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

The selection of which three rows to return is nondeterministic. This means that if you run the query again, without the underlying data changing, theoretically you could get a different set of three rows. In practice, the row selection will depend on physical conditions like optimization choices, storage engine choices, data layout, and other factors. If you actually run the query multiple times, as long as those physical conditions don’t change, there’s some likelihood you will keep getting the same results. But it is critical to understand the “physical data independence” principle from the relational model, and remember that at the logical level you do not have a guarantee for repeatable results. Without ordering specification, you should consider the order as being arbitrary, resulting in a nondeterministic row selection.

Even when you do provide ordering specification, it doesn’t mean that the query is deterministic. For example, an earlier TOP query used `orderdate, DESC` as the ordering specification. The `orderdate` column is not unique; therefore, the selection between rows with the same order date is nondeterministic. So what do you do in cases where you must guarantee determinism? There are two options: using WITH TIES or unique ordering.

The WITH TIES option causes ties to be included in the result. Here’s how you apply it to our example:

```
SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Here’s the result I got from this query:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>11077</td>
<td>2015-05-06</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>11076</td>
<td>2015-05-06</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>11075</td>
<td>2015-05-06</td>
<td>68</td>
<td>8</td>
</tr>
<tr>
<td>11074</td>
<td>2015-05-06</td>
<td>73</td>
<td>7</td>
</tr>
</tbody>
</table>

SQL Server filters the three rows with the most recent order dates, plus it includes all other rows that have the same order date as in the last row. As a result, you can get more rows than the number you specified. In this query, you specified you wanted to filter three rows but ended up getting four. What’s interesting to note here is that the row selection is now deterministic, but the presentation order between rows with the same order date is nondeterministic.
A quick puzzle

What is the following query looking for? (Try to figure this out yourself before looking at the answer.)

```sql
SELECT TOP (1) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY ROW_NUMBER() OVER(PARTITION BY custid ORDER BY orderdate DESC, orderid DESC);
```

Answer: This query returns the most recent order for each customer.

The second method to guarantee a deterministic result is to make the ordering specification unique by adding a tiebreaker. For example, you could add `orderid, DESC` as the tiebreaker in our example. This means that, in the case of ties in the order date values, a row with a higher order ID value is preferred to a row with a lower one. Here's our query with the tiebreaker applied:

```sql
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
```

This query generates the following output:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>11077</td>
<td>2015-05-06</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>11076</td>
<td>2015-05-06</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>11075</td>
<td>2015-05-06</td>
<td>68</td>
<td>8</td>
</tr>
</tbody>
</table>

Use of unique ordering makes both the row selection and presentation ordering deterministic. The result set as well as the presentation ordering of the rows are guaranteed to be repeatable so long as the underlying data doesn’t change.

If you have a case where you need to filter a certain number of rows but truly don’t care about order, it could be a good idea to specify `ORDER BY (SELECT NULL)`, like so:

```sql
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY (SELECT NULL);
```

This way, you let everyone know your choice of arbitrary order is intentional, which helps to avoid confusion and doubt.

As a reminder of what I explained in Chapter 1, “Logical query processing,” about the TOP and OFFSET-FETCH filters, presentation order is guaranteed only if the outer query has an ORDER BY clause. For example, in the following query presentation, ordering is not guaranteed:

```sql
SELECT orderid, orderdate, custid, empid
FROM ( SELECT TOP (3) orderid, orderdate, custid, empid
       FROM Sales.Orders
       ORDER BY orderdate DESC, orderid DESC
     ) AS D;
```
To provide a presentation-ordering guarantee, you must specify an ORDER BY clause in the outer query, like so:

```sql
SELECT orderid, orderdate, custid, empid
FROM ( SELECT TOP (3) orderid, orderdate, custid, empid
       FROM Sales.Orders
       ORDER BY orderdate DESC, orderid DESC ) AS D
ORDER BY orderdate DESC, orderid DESC;
```

The **OFFSET-FETCH filter**

The OFFSET-FETCH filter is a standard feature designed similar to TOP but with an extra element. You can specify how many rows you want to skip before specifying how many rows you want to filter.

As you could have guessed, this feature can be handy in implementing paging solutions—that is, returning a result to the user one chunk at a time upon request when the full result set is too long to fit in one screen or web page.

The OFFSET-FETCH filter requires an ORDER BY clause to exist, and it is specified right after it. You start by indicating how many rows to skip in an OFFSET clause, followed by how many rows to filter in a FETCH clause. For example, based on the indicated order, the following query skips the first 50 rows and filters the next 25 rows:

```sql
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

In other words, the query filters rows 51 through 75. In paging terms, assuming a page size of 25 rows, this query returns the third page.

To allow natural declarative language, you can use the keyword FIRST instead of NEXT if you like, though the meaning is the same. Using FIRST could be more intuitive if you’re not skipping any rows. Even if you don’t want to skip any rows, T-SQL still makes it mandatory to specify the OFFSET clause (with 0 ROWS) to avoid parsing ambiguity. Similarly, instead of using the plural form of the keyword ROWS, you can use the singular form ROW in both the OFFSET and the FETCH clauses. This is more natural if you need to skip or filter only one row.

If you’re curious what the purpose of the keyword ONLY is, it means not to include ties. Standard SQL defines the alternative WITH TIES; however, T-SQL doesn’t support it yet. Similarly, standard SQL defines the PERCENT option, but T-SQL doesn’t support it yet either. These two missing options are available with the TOP filter.
As mentioned, the OFFSET-FETCH filter requires an ORDER BY clause. If you want to use arbitrary order, like TOP without an ORDER BY clause, you can use the trick with ORDER BY (SELECT NULL), like so:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY (SELECT NULL)
OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY;
```

The FETCH clause is optional. If you want to skip a certain number of rows but not limit how many rows to return, simply don’t indicate a FETCH clause. For example, the following query skips 50 rows but doesn’t limit the number of returned rows:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 50 ROWS;
```

Concerning presentation ordering, the behavior is the same as with the TOP filter; namely, with OFFSET-FETCH also, presentation ordering is guaranteed only if the outermost query has an ORDER BY clause.

---

**Optimization of filters demonstrated through paging**

So far, I described the logical design aspects of the TOP and OFFSET-FETCH filters. In this section, I’m going to cover optimization aspects. I’ll do so by looking at different paging solutions. I’ll describe two paging solutions using the TOP filter, a solution using the OFFSET-FETCH filter, and a solution using the ROW_NUMBER function.

In all cases, regardless of which filtering option you use for your paging solution, an index on the ordering elements is crucial for good performance. Often you will get good performance even when the index is not a covering one. Curiously, sometimes you will get better performance when the index isn’t covering. I’ll provide the details in the specific implementations.

I’ll use the Orders table from the PerformanceV3 database in my examples. Suppose you need to implement a paging solution returning one page of orders at a time, based on `orderid` as the sort key. The table has a nonclustered index called PK_Orders defined with `orderid` as the key. This index is not a covering one with respect to the paging queries I will demonstrate.

---

**Optimization of TOP**

There are a couple of strategies you can use to implement paging solutions with TOP. One is an anchor-based strategy, and the other is TOP over TOP (nested TOP queries).

The anchor-based strategy allows the user to visit adjacent pages progressively. You define a stored procedure that when given the sort key of the last row from the previous page, returns the next page. Here’s an implementation of such a procedure:
USE PerformanceV3;
IF OBJECT_ID(N'dbo.GetPage', N'P') IS NOT NULL DROP PROC dbo.GetPage;
GO
CREATE PROC dbo.GetPage
    @orderid AS INT = 0, -- anchor sort key
    @pagesize AS BIGINT = 25
AS
SELECT TOP (@pagesize) orderid, orderdate, custid, empid
FROM dbo.Orders
WHERE orderid > @orderid
ORDER BY orderid;
GO

Here I’m assuming that only positive order IDs are supported. Of course, you can implement such an integrity rule with a CHECK constraint. The query uses the WHERE clause to filter only orders with order IDs that are greater than the input anchor sort key. From the remaining rows, using TOP, the query filters the first @pagesize rows based on orderid ordering.

Use the following code to request the first page of orders:
EXEC dbo.GetPage @pagesize = 25;

I got the following result (but yours may vary because of the randomization aspects used in the creation of the sample data):

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2011-01-01</td>
<td>C0000005758</td>
<td>205</td>
</tr>
<tr>
<td>2</td>
<td>2011-01-01</td>
<td>C0000015925</td>
<td>251</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2011-01-01</td>
<td>C0000003541</td>
<td>316</td>
</tr>
<tr>
<td>25</td>
<td>2011-01-01</td>
<td>C0000005636</td>
<td>256</td>
</tr>
</tbody>
</table>

In this example, the last sort key in the first page is 25. Therefore, to request the second page of orders, you pass 25 as the input anchor sort key, like so:
EXEC dbo.GetPage @orderid = 25, @pagesize = 25;

Of course, in practice the last sort key in the first page could be different than 25, but in my sample data the keys start with 1 and are sequential. I got the following result when running this code:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>2011-01-01</td>
<td>C0000017397</td>
<td>332</td>
</tr>
<tr>
<td>27</td>
<td>2011-01-01</td>
<td>C0000012629</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>2011-01-01</td>
<td>C0000016429</td>
<td>53</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>2011-01-01</td>
<td>C0000015415</td>
<td>95</td>
</tr>
<tr>
<td>50</td>
<td>2010-12-06</td>
<td>C0000008667</td>
<td>117</td>
</tr>
</tbody>
</table>

To ask for the third page of orders, you pass 50 as the input sort key in the next page request:
EXEC dbo.GetPage @orderid = 50, @pagesize = 25;
I got the following output for the execution of this code:

<table>
<thead>
<tr>
<th></th>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>2011-01-01</td>
<td>C0000000797</td>
<td>438</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>2011-01-01</td>
<td>C0000015945</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>2011-01-01</td>
<td>C0000013558</td>
<td>364</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>2011-01-01</td>
<td>C0000019720</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>2011-01-01</td>
<td>C0000000807</td>
<td>160</td>
<td></td>
</tr>
</tbody>
</table>

The execution plan for the query is shown in Figure 5-1. I'll assume the inputs represent the last procedure call with the request for the third page.

![Figure 5-1: Plan for TOP with a single anchor sort key.](image_url)

**FIGURE 5-1** Plan for TOP with a single anchor sort key.

I'll describe the execution of this plan based on data flow order (right to left). But keep in mind that the API call order is actually left to right, starting with the root node (SELECT). I'll explain why that's important to remember shortly.

The Index Seek operator performs a seek in the index PK_Orders to the first leaf row that satisfies the Start property of the Seek Predicates property: \( \text{orderid} > @\text{orderid} \). In the third execution of the procedure, \( @\text{orderid} \) is 50. Then the Index Seek operator continues with a range scan in the index leaf based on the seek predicate. Absent a Prefix property of the Seek Predicates property, the range scan normally continues until the tail of the index leaf. However, as mentioned, the internal API call order is done from left to right. The Top operator has a property called Top Expression, which is set in the plan to \( @\text{pagesize} \) (25, in our case). This property tells the Top operator how many rows to request from the Nested Loops operator to its right. In turn, the Nested Loops operator requests the specified number of rows (25, in our case) from the Index Seek operator to its right. For each row returned from the Index Seek Operator, Nested Loops executes the Key Lookup operator to collect the remaining elements from the respective data row. This means that the range scan doesn't proceed beyond the 25th row, and this also means that the Key Lookup operator is executed 25 times.
Not only is the range scan in the Index Seek operator cut short because of TOP's row goal (Top Expression property), the query optimizer needs to adjust the costs of the affected operators based on that row goal. This aspect of optimization is described in detail in an excellent article by Paul White: “Inside the Optimizer: Row Goals In Depth.” The article can be found here: http://sqlblog.com/blogs/paul_white/archive/2010/08/18/inside-the-optimiser-row-goals-in-depth.aspx.

The I/O costs involved in the execution of the query plan are made of the following:

- Seek to the leaf of index: 3 reads (the index has three levels)
- Range scan of 25 rows: 0–1 reads (hundreds of rows fit in a page)
- Nested Loops prefetch used to optimize lookups: 9 reads (measured by disabling prefetch with trace flag 8744)
- 25 key lookups: 75 reads

In total, 87 logical reads were reported for the processing of this query. That's not too bad. Could things be better or worse? Yes on both counts. You could get better performance by creating a covering index. This way, you eliminate the costs of the prefetch and the lookups, resulting in only 3–4 logical reads in total. You could get much worse performance if you don’t have any good index with the sort column as the leading key—not even a noncovering index. This results in a plan that performs a full scan of the data, plus a sort. That's a very expensive plan, especially considering that you pay for it for every page request by the user.

With a single sort key, the WHERE predicate identifying the start of the qualifying range is straightforward: \( \text{orderid} > @\text{orderid} \). With multiple sort keys, it gets a bit trickier. For example, suppose that the sort vector is \((\text{orderdate}, \text{orderid})\), and you get the anchor sort keys \(@\text{orderdate}\) and \(@\text{orderid}\) as inputs to the GetPage procedure. Standard SQL has an elegant solution for this in the form of a feature called row constructor (aka vector expression). Had this feature been implemented in T-SQL, you could have phrased the WHERE predicate as follows: \( (\text{orderdate}, \text{orderid}) > (@\text{orderdate}, @\text{orderid}) \). This also allows good optimization by using a supporting index on the sort keys similar to the optimization of a single sort key. Sadly, T-SQL doesn’t support such a construct yet.

You have two options in terms of how to phrase the predicate. One of them (call it the first predicate form) is the following: \( \text{orderid} \geq @\text{orderid} \text{ AND (orderdate} > @\text{orderdate OR orderid} > @\text{orderid}) \). Another one (call it the second predicate form) looks like this: \( \text{orderdate} = @\text{orderdate AND orderid} > @\text{orderid} \text{ OR orderdate} > @\text{orderdate} \). Both are logically equivalent, but they do get handled differently by the query optimizer. In our case, there's a covering index called idx_od_oid_i_cid_eid defined on the Orders table with the key list \((\text{orderdate}, \text{orderid})\) and the include list \((\text{custid}, \text{empid})\).
Here's the implementation of the stored procedure with the first predicate form:

```sql
IF OBJECT_ID(N'dbo.GetPage', N'P') IS NOT NULL DROP PROC dbo.GetPage;
GO
CREATE PROC dbo.GetPage
    @orderdate AS DATE   = '00010101', -- anchor sort key 1 (orderdate)
    @orderid   AS INT    = 0,          -- anchor sort key 2 (orderid)
    @pagesize  AS BIGINT = 25
AS

SELECT TOP (@pagesize) orderid, orderdate, custid, empid
FROM dbo.Orders
WHERE orderdate >= @orderdate
    AND (orderdate > @orderdate OR orderid > @orderid)
ORDER BY orderdate, orderid;
GO
```

Run the following code to get the first page:

```
EXEC dbo.GetPage @pagesize = 25;
```

I got the following output from this execution:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>310</td>
<td>2010-12-03</td>
<td>C0000014672</td>
<td>218</td>
</tr>
<tr>
<td>330</td>
<td>2010-12-03</td>
<td>C000009594</td>
<td>10</td>
</tr>
<tr>
<td>90</td>
<td>2010-12-04</td>
<td>C000012937</td>
<td>231</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>2010-12-07</td>
<td>C000019961</td>
<td>282</td>
</tr>
<tr>
<td>410</td>
<td>2010-12-07</td>
<td>C00001585</td>
<td>342</td>
</tr>
</tbody>
</table>

Run the following code to get the second page:

```
EXEC dbo.GetPage @orderdate = '20101207', @orderid = 410, @pagesize = 25;
```

I got the following output from this execution:

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1190</td>
<td>2010-12-07</td>
<td>C000004678</td>
<td>465</td>
</tr>
<tr>
<td>1270</td>
<td>2010-12-07</td>
<td>C000015067</td>
<td>376</td>
</tr>
<tr>
<td>1760</td>
<td>2010-12-07</td>
<td>C000009532</td>
<td>104</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2470</td>
<td>2010-12-09</td>
<td>C000008664</td>
<td>205</td>
</tr>
<tr>
<td>2830</td>
<td>2010-12-09</td>
<td>C000010497</td>
<td>221</td>
</tr>
</tbody>
</table>

Run the following code to get the third page:

```
EXEC dbo.GetPage @orderdate = '20101209', @orderid = 2830, @pagesize = 25;
```

I got the following output from this execution:
As for optimization, Figure 5-2 shows the plan I got for the implementation using the first predicate form.

![Figure 5-2 Plan for TOP with multiple anchor sort keys, first predicate form.](image)

Observe that the Start property of the Seek Predicates property is based only on the predicate `orderdate >= @orderdate`. The residual predicate is `orderdate > @orderdate OR orderid > @orderid`. Such optimization could result in some unnecessary work scanning the pages holding the first part of the range with the first qualifying order date with the nonqualifying order IDs—in other words, the rows where `orderdate = @orderdate AND orderid <= @orderid` are going to be scanned even though they need not be returned. How many unnecessary page reads will be performed mainly depends on the density of the leading sort key—`orderdate`, in our case. The denser it is, the more unnecessary work is likely going to happen. In our case, the density of the `orderdate` column is very low (~1/1500); it is so low that the extra work is negligible. But, when the leading sort key is dense, you could get a noticeable improvement by using the second form of the predicate. Here’s an implementation of the stored procedure with the second predicate form:

```sql
IF OBJECT_ID(N'dbo.GetPage', N'P') IS NOT NULL DROP PROC dbo.GetPage;
GO
CREATE PROC dbo.GetPage
@orderdate AS DATE = '00010101', -- anchor sort key 1 (orderdate)
@orderid AS INT = 0, -- anchor sort key 2 (orderid)
@pagesize AS BIGINT = 25
AS
SELECT TOP (@pagesize) orderid, orderdate, custid, empid
FROM dbo.Orders
WHERE (orderdate = @orderdate AND orderid <= @orderid)
OR orderdate > @orderdate
ORDER BY orderdate, orderid;
GO
```

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderdate</th>
<th>custid</th>
<th>empid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3120</td>
<td>2010-12-09</td>
<td>C0000015659</td>
<td>381</td>
</tr>
<tr>
<td>3340</td>
<td>2010-12-09</td>
<td>C0000008708</td>
<td>272</td>
</tr>
<tr>
<td>3620</td>
<td>2010-12-09</td>
<td>C0000009367</td>
<td>312</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2730</td>
<td>2010-12-10</td>
<td>C0000015630</td>
<td>317</td>
</tr>
<tr>
<td>3490</td>
<td>2010-12-10</td>
<td>C0000002887</td>
<td>306</td>
</tr>
</tbody>
</table>
Run the following code to get the third page:

```
EXEC dbo.GetPage @orderdate = '20101209', @orderid = 2830, @pagesize = 25;
```

The query plan for the implementation with the second form of the predicate is shown in Figure 5-3.

![Figure 5-3 Plan for TOP with multiple anchor sort keys, second predicate form.](image)

Observe that there’s no residual predicate, only Seek Predicates. Curiously, there are two seek predicates. Remember that, generally, the range scan performed by an Index Seek operator starts with the first match for Prefix and Start and ends with the last match for Prefix. In our case, one predicate (marked in the plan as [1] Seek Keys…) starts with `orderdate = @orderdate AND orderid > @orderid` and ends with `orderdate = @orderdate`. Another predicate (marked in the plan as [2] Seek Keys…) starts with `orderdate = @orderdate` and has no explicit end. What’s interesting is that during query execution, if Top Expression rows are found by the first seek, the execution of the operator short-circuits before getting to the second. But if the first seek isn’t sufficient, the second will be executed. The fact that in our example the leading sort key (`orderdate`) has low density could mislead you to think that the first predicate form is more efficient. If you test both implementations and compare the number of logical reads, you might see the first one performing 3 or more reads and the second one performing 6 or more reads (when two seeks are used). But if you test the solutions with a dense leading sort key, you will notice a significant difference in favor of the second solution.

There’s another method to using TOP to implement paging. You can think of it as the TOP over TOP, or nested TOP, method. You work with `@pagenum` and `@pagesize` as the inputs to the GetPage procedure. There’s no anchor concept here. You use one query with TOP to filter `@pagenum` * `@pagesize` rows based on the desired order. You define a table expression based on this query (call it `D1`). You use a second query against `D1` with TOP to filter `@pagesize` rows, but in inverse order. You define a table expression based on the second query (call it `D2`). Finally, you write an outer query against `D2` to order the rows in the desired order. Run the following code to implement the GetPage procedure based on this approach:
IF OBJECT_ID(N'dbo.GetPage', N'P') IS NOT NULL DROP PROC dbo.GetPage;
GO
CREATE PROC dbo.GetPage
    @pagenum AS BIGINT = 1,
    @pagesize AS BIGINT = 25
AS

SELECT orderid, orderdate, custid, empid
FROM ( SELECT TOP (@pagesize) *
    FROM ( SELECT TOP (@pagenum * @pagesize) *
        FROM dbo.Orders
        ORDER BY orderid ) AS D1
    ORDER BY orderid DESC ) AS D2
ORDER BY orderid;
GO

Here are three consecutive calls to the procedure requesting the first, second, and third pages:

EXEC dbo.GetPage @pagenum = 1, @pagesize = 25;
EXEC dbo.GetPage @pagenum = 2, @pagesize = 25;
EXEC dbo.GetPage @pagenum = 3, @pagesize = 25;

The plan for the third procedure call is shown in Figure 5-4.

![Figure 5-4 Plan for TOP over TOP.](image)

The plan is not very expensive, but there are three aspects to it that are not optimal when compared to the implementation based on the anchor concept. First, the plan scans the data in the index from the beginning of the leaf until the last qualifying row. This means that there's repetition of work—namely, rescanning portions of the data. For the first page requested by the user, the plan will scan $1 \times \text{@pagesize}$ rows, for the second page it will scan $2 \times \text{@pagesize}$ rows, for the $n$th page it will scan $n \times \text{@pagesize}$ rows. Second, notice that the Key Lookup operator is executed 75 times even though only 25 of the lookups are relevant. Third, there are two Sort operators added to the plan: one reversing the original order to get to the last chunk of rows, and the other reversing it back to the original order to present it like this. For the third page request, the execution of this plan performed 241 logical reads. The greater the number of pages you have, the more work there is.

The benefit of this approach compared to the anchor-based strategy is that you don’t need to deal with collecting the anchor from the result of the last page request, and the user is not limited to navigating only between adjacent pages. For example, the user can start with page 1, request page 5, and so on.
Optimization of OFFSET-FETCH

The optimization of the OFFSET-FETCH filter is similar to that of TOP. Instead of reinventing the wheel by creating an entirely new plan operator, Microsoft decided to enhance the existing Top operator. Remember the Top operator has a property called Top Expression that indicates how many rows to request from the operator to the right and pass to the operator to the left. The enhanced Top operator used to process OFFSET-FETCH has a new property called OffsetExpression that indicates how many rows to request from the operator to the right and not pass to the operator to the left. The OffsetExpression property is processed before the Top Expression property, as you might have guessed.

To show you the optimization of the OFFSET-FETCH filter, I'll use it in the implementation of the GetPage procedure:

```sql
IF OBJECT_ID(N'dbo.GetPage', N'P') IS NOT NULL DROP PROC dbo.GetPage;
GO
CREATE PROC dbo.GetPage
    @pagenum  AS BIGINT = 1,
    @pagesize AS BIGINT = 25
AS

SELECT orderid, orderdate, custid, empid
FROM dbo.Orders
ORDER BY orderid
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;
GO
```

As you can see, OFFSET-FETCH allows a simple and flexible solution that uses the @pagenum and @pagesize inputs. Use the following code to request the first three pages:

```sql
EXEC dbo.GetPage @pagenum = 1, @pagesize = 25;
EXEC dbo.GetPage @pagenum = 2, @pagesize = 25;
EXEC dbo.GetPage @pagenum = 3, @pagesize = 25;
```

Note Remember that under the default isolation level Read Committed, data changes between procedure calls can affect the results you get, causing you to get the same row in different pages or skip some rows. For example, suppose that at point in time T1, you request page 1. You get the rows that according to the paging sort order are positioned 1 through 25. Before you request the next page, at point in time T2, someone adds a new row with a sort key that makes it the 20th row. At point in time T3, you request page 2. You get the rows that, in T1, were positioned 25 through 49 and not 26 through 50. This behavior could be awkward. If you want the entire sequence of page requests to interact with the same state of the data, you need to submit all requests from the same transaction running under the snapshot isolation level.
The plan for the third execution of the procedure is shown in Figure 5-5.

![Plan for OFFSET-FETCH](image)

**FIGURE 5-5** Plan for OFFSET-FETCH.

As you can see in the plan, the Top operator first requests OffsetExpression rows (50, in our example) from the operator to the right and doesn’t pass those to the operator to the left. Then it requests Top Expression rows (25, in our example) from the operator to the right and passes those to the operator to the left. You can see two levels of inefficiency in this plan compared to the plan for the anchor solution. One is that the Index Scan operator ends up scanning 75 rows, but only the last 25 are relevant. This is unavoidable without an input anchor to start after. But the Key Lookup operator is executed 75 times even though, theoretically, the first 50 times could have been avoided. Such logic to avoid applying lookups for the first OffsetExpression rows wasn’t added to the optimizer. The number of logical reads required for the third page request is 241. The farther away the page number you request is, the more lookups the plan applies and the more expensive it is.

Arguably, in paging sessions users don’t get too far. If users don’t find what they are looking for after the first few pages, they usually give up and refine their search. In such cases, the extra work is probably negligible enough to not be a concern. However, the farther you get with the page number you’re after, the more the inefficiency increases. For example, run the following code to request page 1000:

```sql
EXEC dbo.GetPage @pagenum = 1000, @pagesize = 25;
```

This time, the plan involves 25,000 lookups, resulting in a total number of logical reads of 76,644. Unfortunately, because the optimizer doesn’t have logic to avoid the unnecessary lookups, you need to figure this out yourself if it’s important for you to eliminate unnecessary costs. Fortunately, there is a simple trick you can use to achieve this. Have the query with the OFFSET-FETCH filter return only the sort keys. Define a table expression based on this query (call it \( K \), for keys). Then in the outer query,
join K with the underlying table to return all the remaining attributes you need. Here’s the optimized implementation of `GetPage` based on this strategy:

```sql
ALTER PROC dbo.GetPage
    @pagenum AS BIGINT = 1,
    @pagesize AS BIGINT = 25
AS

WITH K AS
    (
        SELECT orderid
        FROM dbo.Orders
        ORDER BY orderid
        OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY
    )
SELECT O.orderid, O.orderdate, O.custid, O.empid
FROM dbo.Orders AS O
INNER JOIN K
    ON O.orderid = K.orderid
ORDER BY O.orderid;
GO
```

Run the following code to get the third page:

```sql
EXEC dbo.GetPage @pagenum = 3, @pagesize = 25;
```

You will get the plan shown in Figure 5-6.

![Plan for OFFSET-FETCH, minimizing lookups.](image)

**FIGURE 5-6** Plan for OFFSET-FETCH, minimizing lookups.

As you can see, the Top operator is used early in the plan to filter the relevant 25 keys. Then only 25 executions of the Index Seek operator are required, plus 25 executions of the Key Lookup operator (because PK_Orders is not a covering index). The total number of logical reads required for the processing of this plan for the third page request was reduced to 153. This doesn’t seem like a dramatic
improvement when compared to the 241 logical reads used in the previous implementation. But try running the procedure with a page that’s farther out, like 1000:

EXEC dbo.GetPage @pagenum = 1000, @pagesize = 25;

The optimized implementation uses only 223 logical reads compared to the 76,644 used in the previous implementation. That’s a big difference!

Curiously, a noncovering index created only on the sort keys, like PK_Orders in our case, can be more efficient for the optimized solution than a covering index. That’s because with shorter rows, more rows fit in a page. So, in cases where you need to skip a substantial number of rows, you get to do so by scanning fewer pages than you would with a covering index. With a noncovering index, you do have the extra cost of the lookups, but the optimized solution reduces the number of lookups to the minimum.

OFFSET TO | AFTER

As food for thought, if you could change or extend the design of the OFFSET-FETCH filter, what would you do? You might find it useful to support an alternative OFFSET option that is based on an input-anchor sort vector. Imagine syntax such as the following (which shows additions to the standard syntax in bold):

OFFSET { <offset row count> { ROW | ROWS } | { TO | AFTER ( <sort vector> ) } }  
FETCH { FIRST | NEXT } [ { fetch first quantity} ] { ROW | ROWS } { ONLY | WITH TIES }  
[ LAST ROW INTO ( <variables vector> ) ]

You would then use a query such as the following in the GetPage procedure (but don’t try it, because it uses unsupported syntax):

```sql
SELECT orderid, orderdate, custid, empid  
FROM dbo.Orders  
ORDER BY orderdate, orderid  
OFFSET AFTER (@anchor_orderdate, @anchor_orderid) -- input anchor sort keys  
FETCH NEXT @pagesize ROWS ONLY  
LAST ROW INTO (@last_orderdate, @last_orderid); -- outputs for next page request
```

The suggested anchor-based offset has a couple of advantages compared to the existing row count–based offset. The former lends itself to good optimization with an index seek directly to the first matching row in the leaf of a supporting index. Also, by using the former, you can see changes in the data gracefully, unlike with the latter.
Optimization of ROW_NUMBER

Another common solution for paging is using the ROW_NUMBER function to compute row numbers based on the desired sort and then filtering the right range of row numbers based on the input @pagenum and @pagesize.

Here's the implementation of the GetPage procedure based on this strategy:

```sql
IF OBJECT_ID(N'dbo.GetPage', N'P') IS NOT NULL DROP PROC dbo.GetPage;
GO
CREATE PROC dbo.GetPage
    @pagenum  AS BIGINT = 1,
    @pagesize AS BIGINT = 25
AS
    WITH C AS
    (SELECT orderid, orderdate, custid, empid,
        ROW_NUMBER() OVER(ORDER BY orderid) AS rn
        FROM dbo.Orders)
    SELECT orderid, orderdate, custid, empid
    FROM C
    WHERE rn BETWEEN (@pagenum - 1) * @pagesize + 1 AND @pagenum * @pagesize
    ORDER BY rn; -- if order by orderid get sort in plan
GO
```

Run the following code to request the first three pages:

```sql
EXEC dbo.GetPage @pagenum = 1, @pagesize = 25;
EXEC dbo.GetPage @pagenum = 2, @pagesize = 25;
EXEC dbo.GetPage @pagenum = 3, @pagesize = 25;
```

The plan for the third page request is shown in Figure 5-7.

**FIGURE 5-7** Plan for ROW_NUMBER.

Interestingly, the optimization of this solution is similar to that of the solution based on the OFFSET-FETCH filter. You will find the same inefficiencies, including the unnecessary lookups. As a result, the costs are virtually the same. For the third page request, the number of logical reads is 241. Run the procedure again asking for page 1000:

```sql
EXEC dbo.GetPage @pagenum = 1000, @pagesize = 25;
```
The number of logical reads is now 76,644. You can avoid the unnecessary lookups by applying the same optimization principle used in the improved OFFSET-FETCH solution, like so:

```
ALTER PROC dbo.GetPage
    @pagenum  AS BIGINT = 1,
    @pagesize AS BIGINT = 25
AS

WITH C AS
    (
        SELECT orderid, ROW_NUMBER() OVER(ORDER BY orderid) AS rn
        FROM dbo.Orders
    ),
    K AS
    (SELECT orderid, rn
     FROM C
     WHERE rn BETWEEN (@pagenum - 1) * @pagesize + 1 AND @pagenum * @pagesize)
SELECT O.orderid, O.orderdate, O.custid, O.empid
FROM dbo.Orders AS O
INNER JOIN K
    ON O.orderid = K.orderid
ORDER BY K.rn;
GO
```

Run the procedure again requesting the third page:

```
EXEC dbo.GetPage @pagenum = 3, @pagesize = 25;
```

The plan for the optimized solution is shown in Figure 5-8.

```
FIGURE 5-8 Plan for ROW_NUMBER, minimizing lookups.
```

Observe that the Top operator filters the first 75 rows, and then the Filter operator filters the last 25, before applying the seeks and the lookups. As a result, the seeks and lookups are executed only 25 times. The execution of the plan for the third page request involves 153 logical reads, compared to 241 required by the previous solution.

Run the procedure again, this time requesting page 1000:

```
EXEC dbo.GetPage @pagenum = 1000, @pagesize = 25;
```
This execution requires only 223 logical reads, compared to 76,644 required by the previous solution.

Using the TOP option with modifications

T-SQL supports using the TOP filter with modification statements. This section first describes this capability, and then its limitation and a workaround for the limitation. Then it describes a practical use case for this capability when you need to delete a large number of rows.

In my examples, I’ll use a table called MyOrders. Run the following code to create this table as an initial copy of the Orders table in the PerformanceV3 database:

```
USE PerformanceV3;
IF OBJECT_ID(N'dbo.MyOrders', N'U') IS NOT NULL DROP TABLE dbo.MyOrders;
GO
SELECT * INTO dbo.MyOrders FROM dbo.Orders;
CREATE UNIQUE CLUSTERED INDEX idx_od_oid ON dbo.MyOrders(orderdate, orderid);
```

TOP with modifications

With T-SQL, you can use TOP with modification statements. Those statements are INSERT TOP, DELETE TOP, UPDATE TOP, and MERGE TOP. This means the statement will stop modifying rows once the requested number of rows are affected. For example, the following statement deletes 50 rows from the table MyOrders:

```
DELETE TOP (50) FROM dbo.MyOrders;
```

When you use TOP in a SELECT statement, you can control which rows get chosen using the ORDER BY clause. But modification statements don’t have an ORDER BY clause. This means you can indicate how many rows you want to modify, but not based on what order—at least, not directly. So the preceding statement deletes 50 rows, but you cannot control which 50 rows get deleted. You should consider the order as being arbitrary. In practice, it depends on optimization and data layout.

This limitation is a result of the design choice that the TOP ordering is to be defined by the traditional ORDER BY clause. The traditional ORDER BY clause was originally designed to define presentation order, and it is available only to the SELECT statement. Had the design of TOP been different, with its own ordering specification that is not related to presentation ordering, it would have been natural to use also with modification statements. Here’s an example for what such a design might have looked like (but don’t run the code, because this syntax isn’t supported):

```
DELETE TOP (50) OVER(ORDER BY orderdate, orderid) FROM dbo.MyOrders;
```

Fortunately, when you do need to control which rows get chosen, you can use a simple trick as a workaround. Use a SELECT query with a TOP filter and an ORDER BY clause. Define a table expression based on that query. Then issue the modification against the table expression, like so:
WITH C AS
(  
    SELECT TOP (50) *
    FROM dbo.MyOrders
    ORDER BY orderdate, orderid
)
DELETE FROM C;

In practice, the rows from the underlying table will be affected. You can think of the modification as being defined through the table expression.

The OFFSET-FETCH filter is not supported directly with modification statements, but you can use a similar trick like the one with TOP.

Modifying in chunks

Having TOP supported by modification statements without the ability to indicate order might seem futile, but there is a practical use case involving modifying large volumes of data. As an example, suppose you need to delete all rows from the MyOrders table where the order date is before 2013. The table in our example is fairly small, having about 1,000,000 rows. But imagine there were 100,000,000 rows, and the number of rows to delete was about 50,000,000. If the table was partitioned (say, by year), things would be easy and highly efficient. You switch a partition out to a staging table and then drop the staging table. However, what if the table is currently not partitioned? The intuitive thing to do is to issue a simple DELETE statement to do the job as a single transaction, like so (but don’t run this statement):

DELETE FROM dbo.MyOrders WHERE orderdate < '20130101';

Such an approach can get you into trouble in a number of ways.

A DELETE statement is fully logged, unlike DROP TABLE and TRUNCATE TABLE statements. Log writes are sequential; therefore, log-intensive operations tend to be slow and hard to optimize beyond a certain point. For example, deleting 50,000,000 rows can take many minutes to finish.

There’s a section in the log considered to be the active portion, starting with the oldest open transaction and ending with the current pointer in the log. The active portion cannot be recycled. So when you have a long-running transaction, it can cause the transaction log to expand, sometimes well beyond its typical size for your database. This can be an issue if you have limited disk space.

Modification statements acquire exclusive locks on the modified resources (row or page locks, as decided by SQL Server dynamically), and exclusive locks are held until the transaction finishes. Each lock is represented by a memory structure that is approximately 100 bytes in size. Acquiring a large number of locks has two main drawbacks. For one, it requires large amounts of memory. Second, it takes time to allocate the memory structures, which adds to the time it takes the transaction to complete. To reduce the memory footprint and allow a faster process, SQL Server will attempt to escalate from the initial granularity of locks (row or page) to a table lock (or partition, if configured). The first trigger for SQL Server to attempt escalation is when the same transaction reaches 5,000 locks against the same object. If unsuccessful (for example, another transaction is holding locks that the escalated
lock would be in conflict with), SQL Server will keep trying to escalate every additional 1,250 locks. When escalation succeeds, the transaction locks the entire table (or partition) until it finishes. This behavior can cause concurrency problems.

If you try to terminate such a large modification that is in progress, you will face the consequences of a rollback. If the transaction was already running for a while, it will take a while for the rollback to finish—typically, more than the original work.

To avoid the aforementioned problems, the recommended approach to apply a large modification is to do it in chunks. For our purge process, you can run a loop that executes a DELETE TOP statement repeatedly until all qualifying rows are deleted. You want to make sure that the chunk size is not too small so that the process will not take forever, but you want it to be small enough not to trigger lock escalation. The tricky part is figuring out the chunk size. It takes 5,000 locks before SQL Server attempts escalation, but how does this translate to the number of rows you’re deleting? SQL Server could decide to use row or page locks initially, plus when you delete rows from a table, SQL Server deletes rows from the indexes that are defined on the table. So it’s hard to predict what the ideal number of rows is without testing.

A simple solution could be to test different numbers while running a trace or an extended events session with a lock-escalation event. For example, I ran the following extended events session with a Live Data window open, while issuing DELETE TOP statements from a session with session ID 53:

```
CREATE EVENT SESSION [Lock_Escalation] ON SERVER
ADD EVENT sqlserver.lock_escalation(
    WHERE ([sqlserver].[session_id]=(53)));
```

I started with 10,000 rows using the following statement:

```
DELETE TOP (10000) FROM dbo.MyOrders WHERE orderdate < '20130101';
```

Then I adjusted the number, increasing or decreasing it depending on whether an escalation event took place or not. In my case, the first point where escalation happened was somewhere between 6,050 and 6,100 rows. Once you find it, you don’t want to use that point minus 1. For example, if you add indexes later on, the point will become lower. To be on the safe side, I take the number that I find in my testing and divide it by two. This should leave enough room for the future addition of indexes. Of course, it’s worthwhile to retest from time to time to see if the number needs to be adjusted.

Once you have the chunk size determined (say, 3,000), you implement the purge process as a loop that deletes one chunk of rows at a time using a DELETE TOP statement, like so:

```
SET NOCOUNT ON;

WHILE 1 = 1
BEGIN
    DELETE TOP (3000) FROM dbo.MyOrders WHERE orderdate < '20130101';
    IF @ROWCOUNT < 3000 BREAK;
END
```
The code uses an infinite loop. Every execution of the DELETE TOP statement deletes up to 3,000 rows and commits. As soon as the number of affected rows is lower than 3,000, you know that you’ve reached the last chunk, so the code breaks from the loop. If this process is running (and during peak hours), you want to abort it, and it’s quite safe to stop it. Only the current chunk will undergo a rollback. You can then run it again in the next window you have for this and the process will simply pick up where it left off.

**Top N per group**

The *top N per group* task is a classic task that appears in many shapes in practice. Examples include the following: “Return the latest price for each security,” “Return the employee who handled the most orders for each region,” ”Return the three most recent orders for each customer,” and so on. Interestingly, like with many other examples in T-SQL, it’s not like there’s one solution that is considered the most efficient in all cases. Different solutions work best in different circumstances. For *top N per group* tasks, two main factors determine which solution is most efficient: the availability of a supporting index and the density of the partitioning (group) column.

The task I will use to demonstrate the different solutions is returning the three most recent orders for each customer from the Sales.Orders table in the TSQLV3 database. In any *top N per group* task, you need to identify the elements involved: partitioning, ordering, and covering. The partitioning element defines the groups. The ordering element defines the order—based on which, you filter the first N rows in each group. The covering element simply represents the rest of the columns you need to return. Here are the elements in our sample task:

- **Partitioning:** custid
- **Ordering:** orderdate DESC, orderid DESC
- **Covering:** empid

As mentioned, one of the important factors contributing to the efficiency of solutions is the availability of a supporting index. The recommended index is based on a pattern I like to think of as POC—the acronym for the elements involved (partitioning, ordering, and covering). The PO elements should form the index key list, and the C element should form the index include list. If the index is clustered, only the key list is relevant; all the rest of the columns are included in the leaf row, anyway. Run the following code to create the POC index for our sample task:

```sql
USE TSQLV3;
CREATE UNIQUE INDEX idx_poc ON Sales.Orders(custid, orderdate DESC, orderid DESC) INCLUDE(empid);
```

The other important factor in determining which solution is most efficient is the density of the partitioning element (custid, in our case). The Sales.Orders table in our example is very small, but imagine the same structure with a larger volume of data—say, 10,000,000 rows. The row size in our index is
quite small (22 bytes), so over 300 rows fit in a page. This means the index will have about 30,000 pages in its leaf level and will be three levels deep. I'll discuss two density scenarios in my examples:

- Low density: 1,000,000 customers, with 10 orders per customer on average
- High density: 10 customers, with 1,000,000 orders per customer on average

I'll start with a solution based on the ROW_NUMBER function that is the most efficient in the low-density case. I'll continue with a solution based on TOP and APPLY that is the most efficient in the high-density case. Finally, I'll describe a solution based on concatenation that performs better than the others when a POC index is not available, regardless of density.

**Solution using ROW_NUMBER**

Two main optimization strategies can be used to carry out our task. One strategy is to perform a seek for each customer in the POC index to the beginning of that customer's section in the index leaf, and then perform a range scan of the three qualifying rows. Another strategy is to perform a single scan of the index leaf and then filter the interesting rows as part of the scan.

The former strategy is not efficient for low density because it involves a large number of seeks. For 1,000,000 customers, it requires 1,000,000 seeks. With three levels in the index, this approach translates to 3,000,000 random reads. Therefore, with low density, the strategy involving a single full scan and a filter is more efficient. From an I/O perspective, it should cost about 30,000 sequential reads.

To achieve the more efficient strategy for low density, you use the ROW_NUMBER function. You write a query that computes row numbers that are partitioned by `custid` and ordered by `ordertype DESC, orderid DESC`. This query is optimized with a single ordered scan of the POC index, as desired. You then define a CTE based on this query and, in the outer query, filter the rows with a row number that is less than or equal to 3. This part adds a Filter operator to the plan. Here's the complete solution:

```sql
WITH C AS

(SELECT
 ROW_NUMBER() OVER(
    PARTITION BY custid,
    ORDER BY ordertype DESC, orderid DESC) AS rownum,
    orderid, ordertype, custid, empid
 FROM Sales.Orders)

SELECT custid, ordertype, orderid, empid
 FROM C
 WHERE rownum <= 3;
```

The execution plan for this solution is shown in Figure 5-9.
As you can see, the majority of the cost of this plan is associated with the ordered scan of the POC index. As mentioned, if the table had 10,000,000 rows, the I/O cost would be about 30,000 sequential reads.

**Solution using TOP and APPLY**

If you have high density (10 customers, with 1,000,000 rows each), the strategy with the index scan is not the most efficient. With a small number of partitions (customers), a plan that performs a seek in the POC index for each partition is much more efficient.

If only a single customer is involved in the task, you can achieve a plan with a seek by using the TOP filter, like so:

```sql
SELECT TOP (3) orderid, orderdate, empid
FROM Sales.Orders
WHERE custid = 1
ORDER BY orderdate DESC, orderid DESC;
```

To apply this logic to each customer, use the APPLY operator with the preceding query against the Customers table, like so:

```sql
SELECT C.custid, A.orderid, A.orderdate, A.empid
FROM Sales.Customers AS C
CROSS APPLY ( SELECT TOP (3) orderid, orderdate, empid
FROM Sales.Orders AS O
WHERE O.custid = C.custid
ORDER BY orderdate DESC, orderid DESC ) AS A;
```

The execution plan for this solution is shown in Figure 5-10.

You get the desired plan for high density. With only 10 customers, this plan requires about 30 logical reads. That’s big savings compared to the cost of the scan strategy, which is 30,000 reads.
TOP OVER

Again, as a thought exercise, if you could change or extend the design of the TOP filter, what would you do? In the existing design, the ordering specification for TOP is based on the underlying query’s ORDER BY clause. An alternative design is for TOP to use its own ordering specification that is separate from the underlying query’s ORDER BY clause. This way, it is clear that the TOP ordering doesn’t provide any presentation-ordering guarantees, plus it would allow you to use a different ordering specification for the TOP filter and for presentation purposes. Furthermore, the TOP syntax could benefit from a partitioning element, in that the filter is applied per partition. Because the OVER clause used with window functions already supports partitioning and ordering specifications, there’s no need to reinvent the wheel. A similar syntax can be used with TOP, like so:

```
TOP ( < expression > ) [ PERCENT ] [ WITH TIES ]
[ OVER( [ PARTITION BY ( < partition by list > ) ] [ ORDER BY ( <order by list> ) ] ) ] ]
```

You then use the following query to request the three most recent orders for each customer (but do not run this query, because it relies on unsupported syntax):

```
SELECT TOP (3) OVER ( PARTITION BY custid ORDER BY orderdate, orderid )
    orderid, orderdate, custid, empid
FROM dbo.Orders
ORDER BY custid, orderdate, orderid;
```

You can find a request to Microsoft to improve the TOP filter as described here in the following link: [http://connect.microsoft.com/SQLServer/feedback/details/254390/over-clause-enhancement-request-top-over](http://connect.microsoft.com/SQLServer/feedback/details/254390/over-clause-enhancement-request-top-over). Implementing such a design in SQL Server shouldn’t cause compatibility issues. SQL Server could assume the original behavior when an OVER clause isn’t present and the new behavior when it is.

Solution using concatenation (a carry-along sort)

Absent a POC index, both solutions I just described become inefficient and have problems scaling. The solution based on the ROW_NUMBER function will require sorting. Sorting has $n \log n$ scaling, becoming more expensive per row as the number of rows increases. The solution with the TOP filter and the APPLY operator might require an Index Spool operator (which involves the creation of an index during the plan) and sorting.

Interestingly, when $N$ equals 1 in the top N per group task and a POC index is absent, there’s a third solution that performs and scales better than the other two.

Make sure you drop the POC index before proceeding:

```
DROP INDEX idx_poc ON Sales.Orders;
```

The third solution is based on the concatenation of elements. It implements a technique you can think of as a carry-along sort. You start by writing a grouped query that groups the rows by the $P$
element (custid). In the SELECT list, you convert the O (orderdate DESC, orderid DESC) and C (empid) elements to strings and concatenate them into one string. What’s important here is to convert the original values into string forms that preserve the original ordering behavior of the values. For example, use leading zeros for integers, use the form YYYYMMDD for dates, and so on. It’s only important to preserve ordering behavior for the O element to filter the right rows. The C element should be added just to return it in the output. You apply the MAX aggregate to the concatenated string. This results in returning one row per customer, with a concatenated string holding the elements from the most recent order. Finally, you define a CTE based on the grouped query, and in the outer query you extract the individual columns from the concatenated string and convert them back to the original types. Here’s the complete solution query:

WITH C AS
(
    SELECT
        custid,
        MAX( (CONVERT(CHAR(8), orderdate, 112)
            + RIGHT('000000000' + CAST(orderid AS VARCHAR(10)), 10)
            + CAST(empid AS CHAR(10)) ) COLLATE Latin1_General_BIN2 ) AS s
    FROM Sales.Orders
    GROUP BY custid
)

SELECT custid,
    CAST( SUBSTRING(s, 1, 8) AS DATE ) AS orderdate,
    CAST( SUBSTRING(s, 9, 10) AS INT ) AS orderid,
    CAST( SUBSTRING(s, 19, 10) AS CHAR(10) ) AS empid
FROM C;

What’s nice about this solution is that it scales much better than the others. With a small input table, the optimizer usually sorts the data and then uses an order-based aggregate (the Stream Aggregate operator). But with a large input table, the optimizer usually uses parallelism, with a local hash-based aggregate for each thread doing the bulk of the work, and a global aggregate that aggregates the local ones. You can see this approach by running the carry-along-sort solution against the Orders table in the PerformanceV3 database:

USE PerformanceV3;

WITH C AS
(
    SELECT
        custid,
        MAX( (CONVERT(CHAR(8), orderdate, 112)
            + RIGHT('000000000' + CAST(orderid AS VARCHAR(10)), 10)
            + CAST(empid AS CHAR(10)) ) COLLATE Latin1_General_BIN2 ) AS s
    FROM dbo.Orders
    GROUP BY custid
)

SELECT custid,
    CAST( SUBSTRING(s, 1, 8) AS DATE ) AS orderdate,
    CAST( SUBSTRING(s, 9, 10) AS INT ) AS orderid,
    CAST( SUBSTRING(s, 19, 10) AS CHAR(10) ) AS empid
FROM C;
The execution plan for this query is shown in Figure 5-11.

**FIGURE 5-11** Plan for a solution using concatenation.

This exercise emphasizes again that there are usually multiple ways to solve any given querying task, and it’s not like one of the solutions is optimal in all cases. In query tuning, different factors are at play, and under different conditions different solutions are optimal.

## Median

Given a set of values, the median is the value below which 50 percent of the values fall. In other words, median is the 50th percentile. Median is such a classic calculation in the statistical analysis of data that many T-SQL solutions were created for it over time. I will focus on three solutions. The first uses the PERCENTILE_CONT window function. The second uses the ROW_NUMBER function. The third uses the OFFSET-FETCH filter and the APPLY operator.

Our calculation of median will be based on the continuous-distribution model. What this translates to is that when you have an odd number of elements involved, you should return the middle element. When you have an even number, you should return the average of the two middle elements. The alternative to the continuous model is the discrete model, which requires the returned value to be an existing value in the input set.

In my examples, I’ll use a table called T1 with groups represented by a column called `grp` and values represented by a column called `val`. You’re supposed to compute the median value for each group. The optimal index for median solutions is one defined on `(grp, val)` as the key elements. Use the following code to create the table and fill it with a small set of sample data to verify the validity of the solutions:

```sql
USE tempdb;
IF OBJECT_ID(N'dbo.T1', N'U') IS NOT NULL DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
    id  INT NOT NULL IDENTITY
    CONSTRAINT PK_T1 PRIMARY KEY,
    grp INT NOT NULL,
    val INT NOT NULL
);

CREATE INDEX idx_grp_val ON dbo.T1(grp, val);

INSERT INTO dbo.T1(grp, val)
VALUES(1, 30),(1, 10),(1, 100),
      (2, 65),(2, 60),(2, 65),(2, 10);
```
Use the following code to populate the table with a large set of sample data (10 groups, with 1,000,000 rows each) to check the performance of the solutions:

```sql
DECLARE
@numgroups AS INT = 10,
@rowspergroup AS INT = 1000000;

TRUNCATE TABLE dbo.T1;
DROP INDEX idx_grp_val ON dbo.T1;

INSERT INTO dbo.T1 WITH(TABLOCK) (grp, val)
SELECT G.n, ABS(CHECKSUM(NEWID())) % 10000000
FROM TSQLV3.dbo.GetNums(1, @numgroups) AS G
CROSS JOIN TSQLV3.dbo.GetNums(1, @rowspergroup) AS R;

CREATE INDEX idx_grp_val ON dbo.T1(grp, val);
```

**Solution using PERCENTILE_CONT**

Starting with SQL Server 2012, T-SQL supports window functions to compute percentiles. PERCENTILE_CONT implements the continuous model, and PERCENTILE_DISC implements the discrete model. The functions are not implemented as grouped, ordered set functions; rather, they are implemented as window functions. This means that instead of grouping the rows by the `grp` column, you will define the window partition based on `grp`. Consequently, the function will return the same result for all rows in the same partition instead of once per group. To get the result only once per group, you need to apply the DISTINCT option. Here’s the solution to compute median using the continuous model with PERCENTILE_CONT:

```sql
SELECT
    DISTINCT grp, PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY val) OVER(PARTITION BY grp) AS median
FROM dbo.T1;
```

After you overcome the awkwardness in using a window function instead of a grouped one, you might find the solution agreeable because of its simplicity and brevity. That’s until you actually run it and look at its execution plan (which is not for the faint of heart). The plan for the solution is very long and inefficient. It does two rounds of spooling the data in work tables, reading each spool twice—once to get the detail and once to compute aggregates. It took the solution 79 seconds to complete in my system against the big set of sample data. If good performance is important to you, you should consider other solutions.

**Solution using ROW_NUMBER**

The second solution defines two CTEs. One called Counts is based on a grouped query that computes the count (column `cnt`) of rows per group. Another is called RowNums, and it computes row numbers (column `n`) for the detail rows. The outer query joins Counts with RowNums, and it filters only the relevant values for the median calculation. (Keep in mind that the relevant values are those where `n` is `(cnt+1)/2` or `(cnt+2)/2`, using integer division.) Finally, the outer query groups the remaining rows
by the `grp` column and computes the average of the `val` column as the median. Here's the complete solution:

```sql
WITH Counts AS
    (SELECT grp, COUNT(*) AS cnt
     FROM dbo.T1
     GROUP BY grp
    ),
RowNums AS
    (SELECT grp, val,
         ROW_NUMBER() OVER(PARTITION BY grp ORDER BY val) AS n
     FROM dbo.T1
    )
SELECT C.grp, AVG(1. * R.val) AS median
FROM Counts AS C
INNER MERGE JOIN RowNums AS R
    on C.grp = R.grp
WHERE R.n IN ( ( C.cnt + 1 ) / 2, ( C.cnt + 2 ) / 2 )
GROUP BY C.grp;
```

The plan for this solution is shown in Figure 5-12.

**FIGURE 5-12** Plan for a solution using ROW_NUMBER.

SQL Server chose a serial plan that performs two scans of the index, a couple of order-based aggregates, a computation of row numbers, and a merge join. Compared to the previous solution with the PERCENTILE_CONT function, the new solution is quite efficient. It took only 8 seconds to complete in my system. Still, perhaps there’s room for further improvements. For example, you could try to come up with a solution that uses a parallel plan, you could try to reduce the number of pages that need to be scanned, and you could try to eliminate the fairly expensive merge join.

**Solution using OFFSET-FETCH and APPLY**

The third solution I’ll present uses the APPLY operator and the OFFSET-FETCH filter. The solution defines a CTE called `C`, which is based on a grouped query that computes for each group parameters for the OFFSET and FETCH clauses based on the group count. The offset value (call it `ov`) is computed as `(count - 1) / 2`, and the fetch value (call it `fv`) is computed as `2 - count % 2`. For example, if you have a group with 11 rows, `ov` is 5 and `fv` is 1. For a group with 12 rows, `ov` is 5 and `fv` is 2. The outer query applies to each group in `C` an OFFSET-FETCH query that retrieves the relevant values. Finally, the outer query groups the remaining rows and computes the average of the values as the median.
WITH C AS
(  
SELECT grp,  
    COUNT(*) AS cnt,  
    (COUNT(*) - 1) / 2 AS ov,  
    2 - COUNT(*) % 2 AS fv  
FROM dbo.T1  
GROUP BY grp
)

SELECT grp, AVG(1. * val) AS median  
FROM C  
CROSS APPLY (  
    SELECT val  
    FROM dbo.T1 AS O  
    WHERE O.grp = C.grp  
    ORDER BY O.val  
    OFFSET C.ov ROWS FETCH NEXT C.fv ROWS ONLY ) AS A  
GROUP BY grp;

The plan for this solution is shown in Figure 5-13.

The plan for the third solution has three advantages over the plan for the second. One is that it uses parallelism efficiently. The second is that the index is scanned in total one and a half times rather than two. The cost of the seeks is negligible here because the data has dense groups. The third advantage is that there’s no merge join in the plan. It took only 1 second for this solution to complete on my system. That’s quite impressive for 10,000,000 rows.

**Conclusion**

This chapter covered the TOP and OFFSET-FETCH filters. It started with a logical description of the filters. It then continued with optimization aspects demonstrated through paging solutions. You saw how critical it is to have an index on the sort columns to get good performance. Even with an index, I gave recommendations how to alter the solutions to avoid unnecessary lookups.

The chapter also covered modifying data with the TOP filter. Finally, the chapter concluded by demonstrating the use of TOP and OFFSET-FETCH to solve common tasks like top N per group and using a median value.
This page intentionally left blank
Symbols

= (equality) operator
  cardinality estimates for, 110, 113–114, 561
  equi joins, 230–231
> (greater than) operator, cardinality estimates for, 561
< (less than) operator, cardinality estimates for, 561

A

absolute frequencies, 476–479
absolute percentages, 476–479
absolute values, computing, 323–324
access methods.
  See also scans; seeks
  allocation order scan safety issues, 65–76
  allocation order scans vs. index order scans, 65
  clustered index seek + range scan, 93–94
  cost associations of, 57
  covering nonclustered index seek + range scan, 94–97
  dm_db_index_usage_stats function, 135–136
  index order scan issues. See index order scans index seek, 49. See also Index Seek operator
  logical reads as performance metric for, 57
  nonclustered index seek + range scan + lookups, 81–90
  ordered clustered index scans, 62–63
  ordered covering nonclustered index scans, 63–65
  Read Uncommitted isolation level, 68–69, 78, 81
  table scan/unordered clustered index scan, 57–60
  unordered covering nonclustered index scans, 60–62
  unordered nonclustered index scans + lookups, 91–93
acyclic graphs, 708. See also directed acyclic graphs (DAGs)
add outer rows logical processing phase, 4–5, 13
ADO.NET context connection strings with SQLCLR, 603–605
Affinity Mask setting, 175, 177
AFTER DDL triggers, 579–581
AFTER DML triggers, 575–578
AFTER UPDATE triggers, 575–578
aggregate calculations, custom
  carry-along-sort solutions, 326–327
  cursors for, 314–315
  FOR XML string concatenation, 317–319
  hierarchical levels of aggregation. See grouping sets modes, 324–327
  overview of, 313–314
  pivoting for, 315–316, 318–319
  products, 322–324
  SELECT @local_variable method, 319–322
  user-defined. See SQLCLR user-defined aggregates
aggregate functions
  AVG. See averages, AVG window function
  cardinality overestimations, effects of, 100
  cardinality underestimations, effects of, 99
  COUNT. See COUNT function
  cumulative aggregate function plans, 274–275
  expected frequency calculations with, 504–505
  limitations of, 261–262
  parallelism inhibited by certain, 179
  scalar aggregates, 179
  subquery inputs prohibited, 17
  SUM. See SUM function; SUM OVER window function
  window functions. See aggregate window functions
aggregate window functions
  packing interval solutions, 466–468
  similarity to grouped functions, 260
aggregation
  custom calculations of. See aggregate calculations, custom
  functions for. See aggregate functions persisting with grouping sets, 334–337
  user-defined. See SQLCLR user-defined aggregates
aliases, column
  ensuring uniqueness in queries with, 204
  inline vs. external, for derived tables, 205
  in ORDER BY clauses, 14, 20
  referencing, 8, 17
  reuse with APPLY operator, 222–224
ALL variant of UNION operator, 37–38, 250
all-at-once operations, 17–18
Allen’s interval algebra

Allen’s interval algebra, 452
allocation order scans
  defined, 44
  multiple occurrences of rows problem, 65–67
  skipping rows problem, 67–68
storage engine treatment of, 65–76
storage engine’s choice situations, 59
Table Scan operator processing, 58
  vs. index order scans, 47, 65
allocation units
  pages belonging to, 42
  types of, 44
ALTER INDEX REBUILD command, 48
ALTER TABLE not supported by In-Memory OLTP, 704
ALTER_TABLE events, 579–580
A-Marks, 10
analysis of variance (ANOVA), 505–508, 523
ancestors
  GetAncestor method of HIERARCHYID data type,
    758, 764–765
  materialized path solutions, returning with,
    751–753
  nested sets graph solution, returning with, 786
  parents. See parent nodes
    returning, 730–733
AND operator
  intersection calculations with, 452–455
  interval determinations with, 456
  vs. BETWEEN, 784
ANOVA (analysis of variance), 505–508, 523
ANSI_ set options, 568
ANSI SQL standards, 1–2
anti semi joins, 237–238
AS, SELECT clauses with, 17
ascending keys
  B-trees page splits with, 48
  cardinality estimate problems, 564–568
  statistics problems, 107–110
assemblies, .NET, 587–588
ASSEMBLY keyword, 493–494
assignment SELECT for string concatenation,
  319–322
ATOMIC blocks, In-Memory OLTP, 698
atomicity property of transactions, 633
authentication with users’ credentials, 614
autocommit default mode for transactions, 633
AUTO_CREATE_STATISTICS property, 93, 102, 111
average fragmentation in percent, 47–48
averages
  AVG window function, 274–275
  means. See means
  medians. See medians
  modes. See modes
  moving. See moving average value computations
B
  backward scans, 116–117, 179
  balanced trees, 46
  batch execution mode
    columnstore use of, 128–130
    hash algorithm for joins, 243
  batches, calling vs. execution, 525
  BCL (.NET Base Class Library), 597
  bcp.exe utility, 376, 378
  BEGIN TRAN command, 633–636
  BETWEEN operator
    cardinality estimates with, 112–113
    vs. AND, 784
  BI (business intelligence). See statistics for BI
    (business intelligence)
  bill of materials (BOM) scenario, 711–714
  BINARY function, 326
  binary variables, 474
  bitmaps, query tuning with, 170–171
  blocks. See locks
  BOM (bill of materials) scenario, 711–714
  broadcast parallel distribution scheme, 163
  B-trees
    advantages vs. heaps, 43–44
    ascending key pattern, 48
    balanced trees, 46
    bulk-logging requirements, 376
    BW-Tree structure, 677–679
    clustered indexes structured as, 46–50
    leaf levels, 46–47
    nonclustered index seek + range scan + lookups,
      83–90
    nonclustered indexes structured on, 52–53
    uniquifier columns, 46–47
    unordered clustered index scans of, 57–60
  buffers
    buffer pools, 42
    DROPCLEANBUFFERS, 53
    log buffers, flushing of, 634
  bulk loads of data
    2012 vs. 2014 versions, parallel execution plans,
      174
    BULK INSERT command, 376, 378
BULK statements, MERGE statements using, 410–411
bulk-import tools, list of, 376
eager writes for, 374–375
measuring logging, 377–378
minimal logging, requirements for, 376
OPENROWSET function, 378–380
SELECT INTO for, 173–174, 373–376
bulk rowset provider tool, 378–380
BULK_LOGGED recovery model for transactions, 634
business intelligence. See statistics for BI (business intelligence)
BW-Tree structure, 677–679
C
C#: See SQLCLR programming
  caches
  cleaning before measuring performance, 53
defaults for, determining, 387–390
identity property with, 382
pages in, 42
sequence creation performance issues from, 387–394
temporary table issues from, 147–148
cardinality estimates
cardinality estimator component, 97–98
computed column creation for, 107
disjunction (OR) queries, 106–107
equality operator estimates, 110, 113–114, 561
exponential backoff, 106
filtered index creation for, 107
hardcoded estimates, table of, 561
histogram refresh rate issues, 564–565
histograms, derived from, 103–104
inaccurate, detecting, 98
inaccurate, implications of, 100–101, 557–558
join order issues, 233
legacy vs. 2014 estimators, 98–99, 104–107, 114
LIKE predicate with, 112–115
multiple predicate estimates, 104–107
nonclustered index seek + range scan + lookups, 84–85
operator unknowns, table of estimates for, 110
overestimations, implications of, 100–101
BETWEEN predicate with, 112–113
RECOMPILE option, 110
refresh rate issues, legacy vs. new, 107–110
sniffing, disabling, 110
table variables, 143–146
temporary objects for, 139
trace flag 2389, 109
TVPs, for, 572
underestimations, implications of, 99–100
unknowns in, 110–115
unordered nonclustered index scans + lookups, 92–93
variable sniffing, lack of, 560–564
variables as unknown values, 104, 110
viewing, 98
carry-along-sort solutions, 326–327, 366–368
Cartesian products. See also cross joins
  logical query-processing phase details, 8–9
  as logical query-processing phases, 4
CASE expressions
  PIVOT implicit use of, 30–31
type-conversion errors from, 542–543
case sensitivity
  collision of strings issues, 595
  of method and property names in SQLCLR user-defined types, 621
  .NET SQLCLR issues, 595–597
casting
  CAST with SELECT INTO, 374
  DATE or TIME to DATETIME, 440
CD (coefficient of determination), 498–499
Celko, Joe, 778
centers of distribution, 479–481
CHARINDEX function, 247–248
case sensitivity
  collation of strings issues, 595
  of method and property names in SQLCLR user-defined types, 621
check constraints
  CHECK option, views, 212
  negative logic with, 201
  not supported by In-Memory OLTP, 704
  UNKNOWN values with, 11
CHECKIDENT command, 381
CHECKPOINT command, 53
checkpoint process, 42–43
child nodes. See also descendants
  adding, long paths resulting from, 767
defined, 708
HIERARCHYID children queries, 765
nested sets model of, 778
next level, of, 718
parent-child adjacency lists, converting to
  HIERARCHYID, 771–773
chi-squared tests, 501–505, 523
chunked modification with TOP filters, 361–363
cloud platform. See SQL Database
CLR (Common Language Runtime)
  enabling in a SQL Server instance, 493
HIERARCHYID. See HIERARCHYID data type
SQLCLR. See SQLCLR programming
UDFs (user-defined functions). See SQLCLR programming
UDFs with DataAccessKind.Read, 179
user-defined aggregates, 490–495
Virtual Studio for developing, 589–591
clustered indexes
  ALTER INDEX REBUILD command, 48
  B-tree structure of, 46
  B-trees resulting from, 43–44, 46–50
  clustered index keys with nonclustered indexes
    access, 62
  clustered index seek + range scan access method, 93–94
clustered indexes, continued
CLUSTERED keyword, 43
covering indexes, 93–94
creating, 43
FILLFACTOR option, 48
imports, performance hits with, 44
index order scans, 46–47
leaf levels, 46–47
levels in, determining number of, 49–50
levels of, 48–49
not supported by In-Memory OLTP, 704
ordered clustered index scans, 62–63
root pages, 48
uniquifier columns, 46–47
unordered clustered index scans, 57–60
updateable, columnstore indexes, 128–129
COALESCE function, 316
Codd, Edgar F., 2
coefficient of the variation (CV), 486, 523
coefficient of determination (CD), 498–499, 523
cold caches, 53
collection classes, .NET, 629–630
columns
alias processing order issue with SELECT clauses, 14, 20
changes in, recompiles triggered by, 570–571
changing names and types with RESULTS SETS clauses, 574–575
computed columns not supported for In-Memory OLTP, 703
creation order, 21
dense partitioning issue, 150, 154
name requirement in queries against table expressions, 204
referencing, 8, 17
SQL_VARIANT type, 300
subquery correlations, 189
substitution errors in names in subqueries, 201–202
swapping values of, 18
as variables, 473
columnstore indexes
2012 vs. 2014 capabilities, 125, 128
advantages of, 125–127
batch execution, 128–130
clustered, updateable, columnstore indexes, 128–129
column segments, 126
compression, 125, 129
CPU cost benefits, 128
data organization with, 125–126
data type restrictions, 130
data warehousing as target for, 123
delete bitmaps, 129
deltastores, 129
dictionaries, 126
execution plans, 127
hash algorithm for joins, 243
I/O cost reduction benefit, 127
rebuild table recommendation, 129
rowgroups, 126
rowstore vs. columnstore efficiencies, 123–127
segment elimination, 127
syntax example, 125
tuple mover process, 129
update handling, 129
user views of, 129
COLUMNS_UPDATED function, 577
commands. See specific command names
COMMIT TRAN command, 634–636
commit validation errors, 693–696
Common Language Runtime. See CLR (Common Language Runtime)
common table expressions. See CTEs (common table expressions)
compilations (recompilations), 568–571
composable DML, 417
computed columns not supported for In-Memory OLTP, 703
CONCAT function, 316
concatenation
concatenation with comma separation, 629–632
CONCAT_NULL_YIELDS_NULL set option, 568–570
top N per group task solution using, 366–368
concurrency models
pessimistic vs. optimistic concurrency models, 674
vs. isolation levels, table of, 646
consistency property of transactions, 634
core applications, analysis of variations with, 506–508
constraints
CHECK. See check constraints
DEFAULT with NEXT VALUE FOR, 384
dropping, 385
immediacy of for transactions, 634
kinds not supported by In-Memory OLTP, 704
for memory-optimized tables, 676
unique. See UNIQUE constraints
corect context connection strings, 603–605, 607
CONTEXT_INFO with triggers, 584
contingency tables, 501–505, 523
continuous variables, statistics for
centers of distribution, 479–481
declared, 474
descriptive statistics overview, 479
higher population moments, 487–494
kurtosis, 489–495, 523
means, 479, 481–482, 484, 523
medians. See medians
modes of, 324–327, 479–480, 523
normal distributions, 487–488, 509–512
population moments, 479
ranges of distributions, 482–483
skewness, 479, 488–489, 490–495, 523
spread of distribution, 482–486
standard deviations, 486–487, 490, 498, 511, 523
conversion functions
CAST function. See casting
date and time conversions, 431–432
failures from, 431–432
rounding issues, 447–449
TRY_VERSIONS of, 431–432
CONVERT function for dates and times, 431, 433, 435
correlator threads, 164–165
correlated subqueries, 187, 189–194
correlated tables, 184
correlation coefficients, 498–499, 523
correlation vs. causation, 499
Cost Threshold for Parallelism setting, 175–176
COUNT function
GROUP BY with, 16
HAVING with, 16–17
mode calculations with, 324–327
outer joins issue, 17
phase in logical processing, 36–37
covariance, 495–498, 523
covering indexes
benefits of, 93
clustered indexes as, 93–94
memory optimized indexes are always, 677
stored procedures with, 558
covering nonclustered index seek + range scan, 94–97
CPUs
costs, viewing, 176–177
parallel query execution dependence on number of logical, 89–90
CREATE ASSEMBLY command, 493–494, 587–588, 591
CREATE CLUSTERED INDEX, 43
CREATE TYPE command, 571–573
CREATE_TABLE events, triggers on, 579–580
CROSS APPLY operator
column alias preservation with, 223
cost of, 141–142
logical query-processing phase for, 27–28
MAX OVER window function with, 275–276
multiple queries with, 219–220
Parallel APPLY Pattern, 183–186
unpivoting with, 310–311
cross joins
Cartesian product results of, 224
logical query-processing phases, 4, 8–9
physical-join evaluation order, 232–233
rows, generating large numbers of, 216–217
sample data generation with, 225
subquery optimization with, 226–227
syntax, 224
unpivoting with, 308–310
vs. non-equi joins, 231
cross-container queries, 703
cross-database transactions, not supported by
In-Memory OLTP, 705
csc.exe (C# compiler), 589
CTEs (common table expressions)
ancestors, returning, 730–733
anchor members, 722
anchor/recursive data type matching requirement, 736
cycle detection in graphs, 740–742
descendants, returning, 722–723
expensive work, avoiding repetition of, 140–143
MAXRECURSION hint, 728–729
mode computations with, 324–327
multiple in WITH statements, 208
multiple references to allowed, 208–209
nesting of, 208
performance issues, 209
persistence of results, 140–142
query requirements for, 204–205
recursive, 209–211, 718–719
recursive, parallelism inhibited by, 179
scopes of, 204
shortest-path solutions vs. loops, 792–801
subgraph solution, 722–723, 725
subgraphs with path enumeration, 736
syntax of, 207
temporary objects of, 139–140
topological sorts with, 736–739
troubleshooting multiple, 208
updating data with, 402–403
vs. derived tables, 207–208
CUBE function, 331–333
CUMULATE OVER window function
frequency calculations with, 477
vs. PERCENT_RANK for ranking, 288–289
cumulative F distributions, 506–508
cumulative frequencies, 476–479
cumulative percentages, 476–479
CURRENT ROW delimiter, 269–270, 274, 276–278
CURRENT_TIMESTAMP function, 422
cursors
bad reputation of, 581
custom aggregate calculations with, 314–315
exponential moving average calculations with, 515–518
FAST_FORWARD option, 152
fetching, T-SQL inefficiency, 151–153
iterative solutions with, 151–152
not supported by In-Memory OLTP, 704
returned by ORDER BY clauses, 21–22
string concatenation with, 314–315
syntax example, 151
triggers using, 581–583
CV (coefficient of the variation), 486
CXPacket data structure, 163
cyclic graphs, 708, 715–718, 740–742, 792–801
DAGs

DAGs. See directed acyclic graphs (DAGs)
data caches. See caches
Data Definition Language. See DDL (Data Definition Language)
data integrity, filtered indexes to enforce, 122
Data Manipulation Language. See DML (Data Manipulation Language)
data structures, internal. See internal data structures
data type precedence, 542
data warehousing
  bitmaps for parallel optimization of star join queries, 171
columnstore technology for. See columnstore indexes
ETL processes, isolation level for, 648
grouping sets for persisting aggregates, 328
hash algorithm for joins, 243–244
indexing issues, 125
rowstore technology, inefficiency of, 123–127
slowly changing dimensions type 2, 417
star schema model, 123
data-analysis calculations
  aggregates. See aggregation defined, 259
  filters with, 262–263
  grouping sets with. See grouping sets
  inverse distribution functions, 289–291
  limitations without window functions, 261–263
  medians, calculating, 289–291
  ranking calculations with window functions, 281–285
  scalar aggregate subqueries for, 261–262
  statistics window functions, 288–291
  window functions for. See window functions
  year-to-date (YTD) calculation, 280–281
databases
  assembly (.NET) attachment to, 587
  compatibility levels, setting, 98–99
cross-database transactions, not supported by
    In-Memory OLTP, 705
tempdb, 140, 581
DATEDIFF function, 426–430, 438, 440–445
DATEFIRST option, 436–439
DATEFORMAT option, 434, 436
DATEDIFFPART function, 431, 441
DATENAME function, 415
date-only data with DATETIME, 439–440
DATEPART function, 423–424, 436–438, 449–450
DATETIME2 data type, 419–420, 427, 436, 595
DATETIME2FROMPARTS function, 431
DATETIMEFROMPARTS function, 431
DATETIMEOFFSET data type, 419–420, 423, 426, 436, 595
DATETIMEOFFSETFROMPARTS function, 431
DAY function, 425
daylight saving time, extracting state of, 423–425
daylight saving time issues, 420–421
diff and modulo method for weekdays, 437–438
differences between dates, calculating, 426–430
entry formats, table of, 420
EOMONTH function, 431, 441
extracting parts of dates, 423
filtering SARGability, 445–446
first or last day of a period, finding, 440–441
first or last weekday calculations, 443–445
FORMAT function, 432–433
functions returning current date and time, list of, 422
GETDATE function, 422
GETUTCDATE function, 422
grouping by weeks, 449–459
intersecting intervals, 452–456
interval calculations, 450–471
ISDATE function, 425
IsDaylightSavingTime method, 424
island problems with, 295–299
language-dependent results, 425, 434–437
language-neutral formats, 436
last day of the year calculation, 547–549
last modification tracking, 576
leap seconds, 421–422
literals, issues with, 434–436
maximum concurrent interval problems, 456–465
Microsoft Windows time, 421–422
midnight, issues from rounding near, 447
MONTH function, 425
offsets (UTC), extracting, 423
packing intervals, 466–471
PARSE function for, 431–433, 435
performance issues, 432–433
precision of types, 420
previous or next weekday calculations, 441–443
proleptic Gregorian calendar, use of, 419
query tuning for, 445–446
rounding issues, 447–449
search arguments (SARGs), 445–446
SMALLDATETIME data type, 419–420, 439–440, 448
SMALLDATETIMEFROMPARTS function, 431
SQLCLR type translation issues with, 595
storage requirements, table of, 420
SWITCHOFFSET function, 425–426
SYSDATETIME function, 422, 436–437, 440–441, 448–449
SYSDATETIMEOFFSET function, 421–424
TIME data type, 419–420, 422, 440, 595
time zone issues, 420–421
timefromparts function, 431
time-only data with DATETIME, 439–440
TODATETIMEOFFSET function, 426
TRY_CAST function, 432
TRY_CONVERT function, 431–432
unambiguous formats, table of, 436
UTC compared to, 420–421
weekdays, 423, 436–439, 441–445
WHERE clauses with, 445–446
Windows Time service, 421–422
YEAR function, 425
YYYYMMDD format, 435–436
DAY function, 425
daylight saving time, 420–425
DBCC commands
IND, 70–71
OPTIMIZER_WHATIF, 89–90
SHOW_STATISTICS, 101–102
dbo schema, 620
DDL (Data Definition Language)
In-Memory OLTP surface-area restrictions, 703–704
recompilations, 568
triggers on DDL_DATABASE_LEVEL_EVENTS, 579–581
triggers on DDL_TABLE_EVENTS, 579–580
deadlocks
choosing victims to terminate, 657
DEADLOCK_PRIORITY option, 657
error messages generated by, 657–659
example of generating, 657–658
graphs generated for, 657
indexes, relation to, 658–659
isolation levels, relation to, 660
lengths of transactions, 660
measures to reduce, 658–660
mechanics of, 657
physical resource access order issue, 659
query interop environment, In-Memory OLTP, 691–693
retry logic after, 669–670
single-table deadlocks, 660–662
SQL Server monitoring of, 657
trace flags for, 657
DECIMAL data type, 592–593
deduplication, 399–401
DEFAULT constraints with NEXT VALUE FOR, 384
definite integration, 509–512, 523
degree of parallelism. See DOP (degree of parallelism)
degrees of freedom
in analysis of variance, 505–508
chi-squared critical points, 502
defined, 485, 523
delayed durability, 634, 643–645
DELETE clauses
AFTER DELETE triggers with, 575–578
DELETE FROM statements, 361, 399–400
DELETE TOP filters, 360–363, 400
INSTEAD OF triggers with, 578
in MERGE statements, 405–406
with OUTPUT, 413–414
deleting data
archiving deleted data, 413–414
deduplication, 399–401
DELETE for. See DELETE clauses
SELECT INTO another table method, 400–401
TRUNCATE TABLE statements for, 395–399
Demand parallel distribution scheme, 163, 171–173
dense partitioning issue, 150, 154
DENSE_RANK OVER window function, 282–283, 285, 294–296
density vectors, statistics, 102–103
derived tables
aliasing computed columns requirement, 205
APPLY operator for column alias preservation, 222–224
disadvantages of, 206
execution plans of, 206
external aliasing, 205–206
inline aliasing, 205–206
multiple references to, issue with, 207
nesting issues, 206
persistence of results, 140–142
query requirements for, 204–205
scopes of, 204
syntax for creating, 205
temporary objects of, 139–140
DESC keyword in index definitions, 115–116
DESC option, TOP filters, 344–345
descendants. See also subgraphs
children. See child nodes
CTEs (common table expressions), returning with, 722–723
GetDescendant method of HIERARCHYID data type, 757–759
queries on HIERARCHYID data type, 763–764
descending indexes, 63, 115–118
deviations
mean absolute (MAD), 484
mean squared (MSD), 484
standard, 486–487, 490, 498, 511, 523
dichotomous variables, 474
digraphs. See directed graphs (digraphs)
directed acyclic graphs (DAGs)
  bill of materials (BOM) scenario, 711–714
defined, 708
hierarchies as, 709
sorting, 736–739
topological sorts, 736–739
transitive closure of, 787–792
two table requirement, 711
directed graphs (digraphs)
defined, 707–708
returning subgraphs, 719–729
transitive closure of. See transitive closure
dirty pages
CHECKPOINT forced writing of, 377
eager writes, 140, 174–175
mechanics of, 42–43
dirty reads
defined, 645
Read Uncommitted isolation level, 647
vs. isolation levels, table of, 646
discrete variables, 474
disjunctions not allowed by In-Memory OLTP, 705
DISTINCT clauses
improper uses of, 231–232
NULLs, treatment of, 11
ORDER BY clauses with, 20–22
phase in logical processing, 6, 18–20
semi joins with, 237
distinct predicate not supported, 408
distributions
centers of distribution, 479–481
contingency tables, 501–505
degrees of freedom, 485, 502, 505–508
deviations, mean absolute (MAD), 484
deviations, mean squared (MSD), 484
distribution functions, 509–512
frequency distributions, 476–478
higher population moments, 487–494
inter-quartile ranges, 483
inverse distribution functions, 289–291
kurtosis, 489–495, 523
means of, 479, 481–482, 484, 523
medians of. See medians
modes of, 324–327, 479–480, 523
normal distributions, 487–488, 509–512
ranges of, 482–483
skewness, 479, 488–489, 490–495, 523
spread of distribution, 482–486
standard deviations, 486–487, 490, 498, 511, 523
standard normal distributions, 487–488
variances, 485–486, 505–508
distributor threads, 164–165
divide and conquer algorithms, 733
DLLs in SQLCLR, 587, 591
dm_db_index_operational_stats function, 135
dm_db_index_physical_stats function, 135
dm_db_index_usage_stats function, 135–136
dm_db_missing_index_columns function, 136–137
dm_db_missing_index_details view, 136
dm_db_missing_index_group_stats view, 136
dm_exec_procedure_stats, 138
dm_exec_query_profiles view, 138–139
dm_exec_query_stats view, 137–138
dm_exec_trigger_stats, 138
DMFs (dynamic management functions), 134–139
DML (Data Manipulation Language)
  composable DML, 417
  In-Memory OLTP surface-area restrictions, 704–705
DMVs (dynamic management views), 134–139
DOP (degree of parallelism)
  for costing number, 177
  defined, 159
  execution plan examples, 161–162, 164
  Max Degree of Parallelism setting, 175–177
  viewing for a plan, 164
driver tables, 184
DROP CLEANBUFFERS, 53
duplicate row removal. See DISTINCT clauses
duplicates, deleting, 399–401
durability property of transactions, 634
dynamic filtering, 535–542
dynamic management functions (DMFs), 134–139
dynamic management views (DMVs), 134–139
dynamic pivoting, 530–535
dynamic schema, 300–301
dynamic SQL
  batches of code in, 525
dynamic pivoting, 530–535
dynamic search conditions, 535–542
dynamic sorting, 542–546
EXEC AT command, 529
EXEC command, 525–529
interface support, 525–526, 529–530
performance benefits, 539–540
performance issues, 526
security issues, 526–530, 533, 545
sp_executesql procedures, 529–530, 539–540
tools for building and executing, 525
UDF prohibition of, 546
E
eager writes, 140, 174–175
EAV (entity, attribute, value) model, 300–301
edges
  defined, 707
directed vs. undirected, 707–708
  transitive closure duplicate edge elimination, 789
elements, separating from arrays, 245–249
EMAs (exponential moving averages), 515–518
employee organizational chart scenario, 709–711
end of year calculations, 547–549
entropy calculations, 518–521, 523
EOMONTH function, 431, 441
equality operator (=)
cardinality estimates, hardcoded, 561
equi joins, 230–231
equality predicate access method issues, 95–97
equi joins, 230–231
@@error function, 662–663
ERROR_ functions, 664–666
error handling
commit validation errors, 693–696, 702
ERROR_ functions, list of, 664
RAISERROR command, 664–665
retry logic, 669–670
rollbacks as part of, 668
transactions, errors in, 666–668
TRY-CATCH for. See TRY-CATCH constructs
ERROR_NUMBER function, 664–666
escalation of locks, 641–643
estimators, 512
EVENT SESSION clauses, 132
EVENTDATA function, 579–580
events, routines triggered by. See triggers
EXCEPT operator
EXCEPT ALL implementation, 255–256
for handling NULLs in merges, 407–408
logical query-processing phase for, 38–39
NULLs, treatment of, 11
relational operator characteristics of, 249
returned values of, 255
exception handling for SQLCLR stored procedures, 609–613
Exchange operators, 160–166
exclusive locks (X), 636–637
EXEC AT command, dynamic SQL, 529
EXEC command, dynamic SQL
c ode input to, 525–529
security risks of, 526–529
WITH RESULT SETS clauses, 573–575
EXECUTE AS clauses, 546
execution plans
Bitmap operator in, 170–171
cached plans, set options of, 568–570
cardinality estimates in, viewing, 98–100
CLR scalar functions vs. T-SQL scalar functions, 588–589
clustered index seek + range scan, 93–94
columnstore indexes, 127
covering nonclustered index seek + range scan, 96
CPU costs, viewing, 176–177
cursor-based queries, 152
derived table expressions, 206
descending index plans, 116
Distribute Streams variant of Exchange operator, 160–163
dm_exec_query_profiles view, 138–139
Exchange operators, 160–163
expensive part of, identification, 154
few outer rows optimization, 168–170
filter selectivity, example of plan dependence on, 555–556
Gather Streams variant of Exchange operator, 160–163, 165–166
Hash Match operator, 167
Include Actual Execution Plan option, 138–139
index order, reliance on, 543
Index Scan operators, 65, 168–170
I/O costs, viewing, 176
join algorithm identification in, 239
KEEPFIXED PLAN hint, 570–571
Nested Loops operators, 98, 183–186, 240–241
nonclustered index seek + range scan + lookups, 81–90
NonParallelPlanReason node, 180–181
operator interfaces, 162
OPTIMIZE FOR hint, 561–562
ordered clustered index scans, 62–63
ordered covering nonclustered index scans, 63–65
Ordered property in, 65
Parallel APPLY Pattern, 185
parallel query plans, 159–160
parameter sniffing, 555–558
partitioned tables with demand-based row distribution, 173
physical execution plans, 97
Plan Explorer tool, 139
plan optimality, 568, 570–571
plan stability, 568–570
properties window features, 164
query optimizer, SQL, 2
query revisions effects on, 154
recompilations, 568–571
recursive CTE queries, 211
Repartition Streams variant of Exchange operator, 161
requesting, 53–54
reusable vs. one-time issues, 535–542, 561–562
rowstore technology, 124–125
SARG vs. non-SARG, 446
scalar UDFs, of, 548
Seek Predicates property, 96
SELECT INTO parallel plans, 174
stored procedures, example of plan dependence on filter selectivity, 555
stored procedures, preventing reuse for, 558–560
stored procedures, reuse for, 554–558
strategies, testing for desired, 154–158
Stream Aggregate in, 153, 167
Table Scan operators, 65
table scan/unordered clustered index, 57–58
table variables and recompiles, 145
temporary tables, of, 147
trivial plans, 93
unknowns in cardinality estimates, 110–115
execution plans, continued  
unordered covering nonclustered index scans,  
60–61
unordered nonclustered index scans + lookups,  
91–92 
variables, stored procedure, 561–562
existing value range identification. See island 
problems
EXISTS predicate  
anti semi joins with NOT, 238
identifying gaps problem, 198–200
minimum missing values, finding, 195–200
nested NOT EXISTS, 200–201
NOT EXISTS vs. NOT IN, 204
positive vs. negative logic, 200–201
returns true or false, 194–195
semi joins with, 237
subqueries as inputs to, 194–201
subquery SELECT list index ignored, 195
tuning with, 157
vs. IN predicate, 194–195
EXP function, 322–324
expected frequencies, 501–505
exponential backoff, 106
exponential moving averages (EMAs), 515–518
Extended Events sessions  
creating, 132
extracting statistics from, 132–134
performance, code for viewing, 54
prioritizing queries for tuning, 131–134
query_hash action, 131–134, 137–138
query_post_execution_showplan event, 138
statement completed events, 131–134
Watch Live Data window statistics, 56
extents, 42–43

F
FALSE value, 10
few outer rows optimization, 168–170
filegroup, memory optimized data, 675
FileTable feature with HIERARCHYID type, 754
filtered indexes, 120–122
filters  
access methods for. See access methods
cardinality overestimations, effects of, 100
cardinality underestimations, effects of, 99
with data-analysis calculations, 262–263
date and time SARGability, 445–446
deadlocks resulting from no supporting index,  
658–659
dynamic filtering, 535–542
multiple predicate cardinality estimates, 104–107
multiple predicate issue, 95–97
NULL values, treatment of, 11
selectivity, example of execution plan 
dependence on, 555–556
UNKNOWN values, treatment of, 11
unordered nonclustered index scans + lookups,  
91–92
user-defined, dynamic, 535–542
WHERE-based. See WHERE filters
window functions using, 263–264
first regression line, 500
FIRST_VALUE OVER window function, 285–286, 511
FLOOR function, 468
flow diagram, logical query-processing, 5
fn_dblog function, 377–378
FOLLOWING keyword, 269, 273–274
FOR XML string concatenation, 317–319
FORCE ORDER hint for joins, 233, 236–237, 245
FORCEPLAN set option, 568
foreign-key constraints not supported by In-Memory
OLTP, 704
forests, 708. See also graphs; trees
FORMAT function, 432–433
fragmentation  
allocation vs. index order scans, effects of, 65
average page population, 135
code for checking level of, 70–71
defined, 47–48
dm_db_index_physical_stats function, 135
logical scan fragmentation, 135
rebuilding as remedy, 135
frames, window. See window frames
frequency calculations  
defined, 523
expected frequencies, 501–505
frequency distribution analysis, 476–479
for use in chi-squared calculations, 502–505
with window functions, 477–479
without window functions, 476
FROM clauses  
aliases created in, visibility of, 222–223
derived tables using, 205–207
logical order of, 3–5
logical query-processing phase details, 8–14
multiple operators, processing order of, 14
phase in logical processing, 4–5
sample query for logical phases, 7
table operators in, logical phases of, 26–35
virtual tables generated by, 8
F-tests, 506–508, 523
Full data recovery model  
durability guarantee from, 634
SELECT INTO operations with, 377–378
FULL OUTER JOIN clauses, 13
full table scans when no order required, 57–60

G
GAC (Global Assembly Cache), 587
GAMs (global allocation maps), 42
gaps problems, 291–292
Gather Streams operation, 88, 160–166
Gaussian curves, 487–488
GETDATE function, 422
GETUTCDATE function, 422
global allocation maps (GAMs), 42
Global Assembly Cache (GAC), 587
graphical execution plans. See execution plans
graphs
acyclic, 708. See also directed acyclic graphs (DAGs)
ancestors, returning, 730–733
components of, 707–708
CTE-based solutions. See CTEs (common table expressions)
cycle detection, 740–742
cyclic, 708, 715–718, 792–801
directed, 707–708, 719–729
directed acyclic. See directed acyclic graphs (DAGs)
edges of, 707–708, 789
HIERARCHYID data type for materialized path model. See HIERARCHYID data type
iterative solutions. See iterative graph solutions
materialized paths for solving. See materialized path model
MAXRECURSION hint, 728–729
nested sets solution, 778–786
nodes of. See nodes
paths, returning, 730–733
recursive solutions to, 718–719, 722–723
scenarios used to illustrate, 709–718
sorting, 736–739
subgraphs, algorithm for returning, 719–729
subgraphs with path enumeration, 733–736
transitive closure of directed. See transitive closure
trees as, 708
types of, 707–708
undirected, 708
undirected cyclic. See undirected cyclic graphs
vertexes. See nodes
weighted, 714, 718
greater than (>) operator cardinality estimates, hardcoded, 561
Gregorian calendar, use of, 419
GROUP BY clauses
COUNT with, 16
dates and times, grouping by weeks, 449–459
grouping sets, 15
GROUPING SETS subclauses, 328–331
logical order of, 3–5
logical phase order sample, 7
NULL value treatment, 11
phase in logical processing, 4–5, 15–16
with PIVOT operators, 30
running total calculation with, 272–273
SELECT with, 16
group functions vs. window functions, 259
grouped query data-analysis calculation limitations, 261

H
hashes
cardinality overestimations, effects of, 100
cardinality underestimations, effects of, 99
hash algorithm for joins, 243–245
Hash Join operators in parallel query execution, 170–171
Hash Match operator, 167
hash parallel distribution scheme, 163–165
HAVING filters
COUNT with, 16–17
logical order of, 3–5
logical phase order sample, 7
phase in logical processing, 4–5, 16–17
subquery inputs prohibited, 17
UNKNOWN values, treatment of, 11
headers, page, 42
headers, statistics, 102
heaps
bulk-logging requirements, 376
nonclustered index seek + range scan + lookups, 81–83
nonclustered indexes structured on, 50–51
organization of, 43–46
table scans of, 57–60
Hekaton project. See In-Memory OLTP
hierarchies
defined, 709
employee organizational chart scenario, 709–711
HIERARCHYID for representing. See HIERARCHYID data type
HIERARCHYID data type

HIERARCHYID data type
adding leaf nodes, 756–759
advantages for materialized path model, 754
ancestors queries, 764–765
case sensitivity, 755
children queries, 765
compared to custom graph implementation, 754
conflict prevention, 757–758
data maintenance requirements, 756
DescendantLimit internal method, 763–764
descendants queries, 763–764
execution plans for, 764
GetAncestor method, 758, 765
GetDescendant method, 757–759
GetReparentedValue method, 760–762
GetRoot method, 757
hid attribute, 757–762, 766
index creation for, 755
IP address sorting with, 773–777
IsDescendantOf method, 760, 763
leaf node queries, 765
long paths issue, 767–771
methods provided by, 755
normalizing values of, 767–771
parent-child adjacency lists, converting to
HIERARCHYID, 771–773
path calculations, 757–762
path queries, 764–765
portability issue, 754
presentation queries, 766
querying solutions built with, 758–759, 763–766
sibling positioning, 758
sorting separated lists, 773–777
stored procedures using, 756–757
subgraph queries, 763–764
subtrees, moving, 760–762
table creation using, 755
validity of trees, enforcing, 756
higher population moments, 487–494
histograms
AUTO_CREATE_STATISTICS option, 102
AVG_RANGE_ROWS, 102–104
cardinality estimates from, 103–104
creation scenarios, 102
DISTINCT_RANGE_ROWS, 102–103
EQ_ROWS, 102–103
filtered index, 120–122
frequency of data, 476–479
of index key values, 85–86
of non-index key columns, 93
not maintained for table variables, 553
RANGE_HI_KEY, 102–104
RANGE_ROWS, 102–103
refresh rate issues, 107–110, 564–565
steps in, 103
trace flag 2389, 109
HOBT (heap or B-tree) organization, 43–44
HOLDLOCK hint, 409
Hyper-Threading, 178

IAMS (index allocation maps), 44–45, 47
identifying gaps problem, 198–200
identity columns, 412–413
identity property
aspects and properties of, 394–395
cache performance issues, 387–394
caches with, 382–383
IDENTITY function, 374
limitations of, 381–382
manual value changes, 381
missing values, causes of, 387
performance considerations, 387–394
SELECT INTO copying of, 374
sequence objects, compared to, 390–395
surrogate key generation with, 381–382
trace flag 272, 382, 387
TRUNCATE TABLE, preserving despite, 396–397
identity values, capturing multiple, 412–413
I-Marks, 10
implicit APPLY operator, 221–222
IMPLICIT_TRANSACTIONS option, 633
importing data. See bulk loads of data
IN disjunctions not allowed by In-Memory OLTP, 705
IN predicate vs. EXISTS predicate, 194–195
INCLUDE clauses, 119
included non-key columns, 119–120
inconsistent analysis, 645
INDEX <index_name> CLUSTERED, 43
index allocation maps (IAMS), 44–45, 47
index order scans
defined, 46
safety issues with, 76–81
storage engine choice situations, 59
vs. allocation order scans, 47, 65
Index Scan operators, 65, 168–170
index seek access method, 49
Index Seek operator. See also seeks
forcing as part of a strategy, 154–155
index seek + range scan + lookups access
method, 81–82
inefficiency in Max Concurrent Intervals task, 459
optimizing with subqueries, 192–193
parallel scans with, 168
TOP filters with various indexes, optimization of, 348–352
indexes
ALTER INDEX REBUILD command, 48
clustered. See clustered indexes
columnstore. See columnstore indexes
covering. See covering indexes
deadlocks from lack of, 658–659
IN_ROW_DATA allocation units

descending, 63, 115–118
dm_db_index_operational_stats function, 135
dm_db_index_usage_stats function, 135–136
dm_exec_query_stats view, 137–138
filtered, 120–122
frequency of use, determining, 135–136
imports, performance hits with, 44
INCLUDE clauses in, column order issues, 95–96
included non-key columns, 119–120
INDEXPROPERTY function, 49
inline index definition, 130–131
join nested loops, for, 239–240
key updates as a source of row return errors, 76–81
leaf updates as a source of row return errors, 76–81
levels in, determining number of, 49–50
memory-optimized tables. See indexes for memory-optimized tables
missing index information objects, 136
nonclustered. See nonclustered indexes
page data structure for, 42
POC (partitioning, ordering, covering) pattern, 192, 271–273, 284–285
REBUILD keyword, 135
REORGANIZE keyword, 135
root pages, 48
statistics created with, 85–86, 101
TOP filters, effects on optimization of, 349
UDF CLR functions, setting on, 590
WHERE clause predicate, 120
indexes for memory-optimized tables
BW-Tree structure, 677–679
clustered vs. nonclustered not meaningful, 677
covering nature of, 677, 680
hash indexes, 680–690
key values, 679
latch-free nature of, 679
leaf pages, 679
must be specified in CREATE TABLE statements, 675
nonclustered indexes, 677–680
page-mapping tables, 678
payloads, 679
pointer-based nature of, 676
single-direction nature of, 679–680
syntax for creating nonclustered, 677
information theory, 518–521
inhibitors of parallelism, 178–181
inline index definition, 130–131
inline TVFs (table-valued functions)
end of year calculation example, 548–549
input parameter support, 215
integer sequence generator, 215–218
query requirements for, 204–205
scopes of, 204
syntax for, 215
In-Memory OLTP
2014 introduction of, 671
ALTER TABLE not supported by, 704
ATOMIC blocks, 698
blocking, 691–693
check constraints not supported, 704
clustered indexes not supported by, 704
commit validation errors, 693–696, 702
computed columns not supported, 703
constraints not supported, types of, 704
cross-container queries, 703
cross-database query execution bar, 696
cursors not supported, 704
data always in memory, 672–673
deadlocks, 691–693
DURABILITY option, 675
execution engine efficiency, 673
execution environment for, 690
foreign-key constraints not supported, 704
indexes for. See indexes for memory-optimized tables
integration with SQL Server, 674
isolation levels in query interop, 691
isolation semantics for transactions, 690
LOB data types not supported, 704
lock and latch-free architecture of, 673–674
memory optimized data filegroup, 675
MEMORY_OPTIMIZED option, 675
memory-optimized tables, 672
modern computing environment for, 671–672
natively compiled stored procedures, 673
natively compiled procedures. See natively compiled procedures environment,
In-Memory OLTP
optimistic concurrency model, 674
page elimination, 672–673
parallelism not used currently, 696–697
query interop environment, 690–698
schema options, 675
statistics generation for optimization, 697–698
surface-area restrictions, 703–705
tables, memory-optimized, 675–676
timestamps, 672
transaction isolation and consistency, 672
unique constraints not supported, 704
inner joins
default type with JOIN keyword, 229
logical steps in, 228
logical-join evaluation order, 234–237
ON clauses mandatory, 228
ON vs. WHERE clauses with, 15
physical-join evaluation order, 232–233
running total calculation with, 272–273
syntax for, 228
WHERE clauses with, 228
inner queries. See subqueries
IN_ROW_DATA allocation units, 44
INSERT clauses

INSERT clauses
  AFTER INSERT triggers with, 575–578, 581–585
  EXEC statements, full logging vs. other bulk methods, 376
  INSTEAD OF INSERT triggers with, 578
  INTO clauses with, 412–413
  in MERGE statements, 405–406
  OUTPUT clauses with, 412–413

INSERT SELECT clauses
  casting column types with, 374
  guaranteed identity value order, 374
  identity property, as alternative to, 381
  minimal logging, requirements for, 376
  OUTPUT clauses with, 417
  SELECT INTO compared to, 374–375

INSERT TOP filters, 360
inserting data, SELECT INTO command for, 140, 173–174, 373–376
instance-level parallelism setting, 175–176
INSTEAD OF DML triggers, 578
integration, definite, 509–512
intent exclusive locks (IX), 637
intent shared locks (IS), 637–639
intercepts, 499–501
internal data structures
  allocation units, 44
  B-trees as. See B-trees
  extents, 42–43
  headers, page, 42
  heaps as. See heaps
  heaps, nonclustered indexes structured on, 50–51
  IAMs (index allocation maps), 44–45, 47
  leaf levels, 46–49
  pages, 42–48
  partitions, 44
  row-offset arrays of pages, 42
  table organization, 43–53
internal nodes, 708
inter-quartile ranges, 483, 523
INTERSECT operator
  characteristics of, 249
  INTERSECT ALL version implementation, 254
  logical query-processing phase for, 38–39
  NULLs, treatment of, 11, 249, 253
  ORDER BY clauses with, 249
  syntax for, 253
intersection interval calculations for dates and times, 452–456
interval calculations for dates and times
  Allen’s interval algebra, 452
  AND operator performance issues, 252–255
  CTE for counting active intervals, 457
  delimiters, open vs. closed, 256
  grouping counted intervals, 458
  index optimization for, 252–255
  intersections, 452–456
  maximum concurrent interval problems, 456–465
  packing intervals, 466–471
parallel treatment for performance, 464–465, 470–471
performance in max concurrences solutions, 458–465
performance vs. simplicity of queries, 450–451
Relational Interval Tree (RI-tree) model, 456
ROW_NUMBER OVER window function for, 461–465
window aggregate function with frame solution to max concurrences task, 459–461
INTERVAL not supported, 276–277
intraquery parallelism. See parallel query execution
inverse distribution window functions, 289–291
I/O costs, viewing, 176
IP address sorting, 773–777
IQRs (inter-quartile ranges), 483, 523
ISDATE function, 425
IsDaylightSavingTime method, 424
island problems
  date/time based, 295–299
  DENSE_RANK queries, 294–296
  existing value range identification equivalence, 291
  ignoring small gaps variation, 296–299
  LAG OVER function for, 296–299
  ROW_NUMBER OVER queries, 293–294
  SUM OVER function for, 296–299
ISNULL function, 323
ISO SQL standards, 1–2
isolation levels
  consistent data control purpose of, 645
cross-container queries, for, 703
deadlocks, relationship to, 660
defaults, 646
dirty reads, 645
isolation models, 646
lost updates, 645
NOLOCK hints, 647–648
nonrepeatable reads, 645
phantoms, 645–646
query interop environment, In-Memory OLTP, 691
query level hints, 646
Read Committed, 646, 648–650, 660
Read Committed Snapshot, 646, 652, 655–656, 660
Read Uncommitted, 646, 647–648
READPAST hint, 646
Repeatable Read, 646, 649–651
Serializable, 409, 646, 651–652
SET TRANSACTION ISOLATION LEVEL statements, 646
Snapshot, 646, 652–655
table of available, 646
isolation property of transactions, 634
iterative graph solutions
  advantages of, 718
  ancestors, returning, 730–733
left outer joins

CTEs for, 718–719, 722–723
logic encapsulation choices, 718–719
loops in, 718, 721, 723–728, 730–733
next levels, 718
recursion vs. loops, 718
stored procedures for, 718–719
subgraphs, algorithm for returning, 719–729
subgraphs with path enumeration, 733–736
traversing by stored edges, 718

iterative solutions
cursor-based solutions, 151–152
inefficiency in T-SQL, 149–150, 152–153
iterative execution model of SQL Server, 149–150
separating elements problem, inefficiency of, 246
strong concatenation using cursors, 314–315
superiority of CLR functions for, 597
UDFs using, 549–550
vs. set-based approaches, 149–153

Joins
algorithms for, 239–245
anti semi, 237–238
cardinality overestimations, effects of, 100
cardinality underestimations, effects of, 99
Cartesian product logical processing phase, 8–9
cross. See Cartesian products; cross joins
default type with JOIN keyword, 229
equi, 230–231
FORCE ORDER hint, 233, 236–237, 245
forcing algorithm strategies for, 244–245
hash algorithm for, 243–245
index support for, 239–240, 243
inner. See inner joins
JOIN operator, elements of, 27
left. See left outer joins
left outer. See left outer joins
logical phase order sample, 7
logical-join evaluation order, 234–237
LOOP hint for nested loops algorithm, 244–245
merge algorithm, 241–243, 244–245
multi-join queries, 231–237
nested loops join algorithm, 239–241, 244–245
Nested Loops performance issue with, 184
non-equi, 230–231
operator for, 27
OPTION clause hints, 244–245
outer. See outer joins
parentheses with multiple, 235–236
physical-join evaluation order, 232–233
prefetches with, 241
self, 230
semi, 237–238
SQL-89 vs. SQL-92 syntax, 224–229
star. See star join queries
suboptimal join order issues, 232–233

K
KKEEPFIXED PLAN hint, 570–571
keys
automating creation with sequence objects, 384–385
columns defined as in B-tree clustered indexes, 46
density, effects on optimization, 351–352
foreign-key constraints not supported by In-Memory OLTP, 704
index seek access method for, 49
key columns added to index implicitly, 102
key lists, restrictions on, 119
lookups in nonclustered indexes, 53
lookups, nonclustered index seek + range scan +, 81–90
primary, 43, 408–409
values, histograms of, 85–86
Krieger, Hans-Peter, 456
kurtosis, 489–495, 523

L
LAG OVER window function, 287, 296–299
languages
LANGUAGE option, 568
SET LANGUAGE command, 434–435
last day of the year calculations, 547–549
LAST_VALUE OVER window function, 285–286, 287, 511
latches, 673–674
lateral derived tables. See APPLY operator
lazywriter process, 42–43
LEAD OVER window function, 287, 291–292
leaf layout, showing with DBCC IND command, 70–71
leaf levels
clustered index, rows stored in, 46–47
ordered clustered index scans of, 62–63
rows, clustered index, 93
leaf nodes
adding into HIERARCHYID graphs, 756–759
shake effects of adding new, 743, 782
vs. internal nodes, 708
leaf pages
BW-Tree structure, 678–679
density, 48–49
rows per, access method dependence, 94–95
splits, 47–48
leap seconds, 421–422
left outer joins. See also outer joins
anti semi joins from, 238
followed by other joins, issues with, 234–235
LEFT JOIN, LEFT OUTER JOIN equivalence, 229
logical phase order sample, 7
preserved tables, 13
left outer joins

left outer joins, continued

self join example, 230

subphase of logical processing of, 8–9

vs. right outer joins, 230

less than (<) operator cardinality estimates, 561
LIKE predicate, cardinality estimates with, 112–115

line dependencies

chi-squared tests, 501–505, 523

coefficient of determination (CD), 498–499, 523

correlation coefficients, 498–499, 523
covariance, 495–498, 523

intercepts, 499–501

linear regression, 499–501, 523

slopes, 499–501
two continuous variable dependencies, 495–500

literals, date and time, 434–436

LOB (Large Object) data types

LOB_DATA allocation units, 44

not supported by In-Memory OLTP, 704

@local_variable method, 319–322

locks

blocking function of, 636
blocking situations, 638–640, 657–658. See also deadlocks
deadlocks resulting from. See deadlocks
delayed durability, 643–645
dm_exec_connections view for troubleshooting, 639–640
dm_exec_requests view for troubleshooting, 640
dm_exec_sessions view for troubleshooting, 640
dm_os_waiting_tasks view for troubleshooting, 640
dm_tran_locks view for troubleshooting, 639

escalation, 361–363, 641–643
exclusive locks (X), 636–637
In-Memory OLTP vs. traditional, 691–693

intent exclusive locks (IX), 637

intent shared locks (IS), 637–639

isolation function of, 636

isolation levels with. See isolation levels

KILL command, 641

LOCK_TIMEOUT option, 638, 641

memory requirements of, 641

NOLOCK hints, 647–648

Read Committed isolation level, 638

Read Uncommitted isolation level, 636–637

READPAST hint, 646

resource hierarchy conflict detection, 637

SELECT INTO, created by, 375

shared locks (S), 636–639, 648–649

shared with intent exclusive locks (SIX), 637

SQL Database alternative model for, 636

TOP modification statement issues from, 361–362
update locks (U), 637

UPDLOCK hints, 637, 650–651

WAIT status, 639

write-ahead logging, 643

logarithms, 322–324

Logical Operation property, 239

logical order vs. typed order, 3

logical processing order vs. physical order, 2–3

logical query-processing phases

add outer rows phase, 4–5, 13

APPLY operator, 26–29

Cartesian product phase, 4–5, 8–9

descriptions of, brief, 4, 6

DISTINCT phase, 6, 18–20
evaluate expressions phase, 6

EXCEPT operator, 38–39

flow diagram of, 5

FROM phase, 4–5, 8–14

GROUP BY phase, 4–5, 15–16

HAVING phase, 4–5, 16–17

INTERSECT operator, 38–39

logical order vs. typed order, 3

OFFSET-FETCH filter, 6, 22–26

ON predicate phase, 4–5, 9, 11–13

ORDER BY, 6, 20–22

order of phases, 3–4

PIVOT operator, 29–31

sample query, 6–8

SELECT phase, 4–5, 17–20

step numbers, 3

table operators, 26–27

TOP filter, 6, 22–26

UNION operator, 38–39

UNPIVOT operator, 27, 31–35

table operators, 26–27

WHERE phase, 4–5, 14–15

window functions, 35–37, 268–269

logical reads

as performance metric for access methods, 57

STATISTICS IO output data, 55

logs

active portion, expansion issues, 361

flushing buffers to complete transactions, 634

log-intensive operations, 361

measuring logging, 377–378

SELECT INTO command logging options, 374

test for statistics during sequence creation, 390–393

write-ahead logging, 643

lookups

expense of, minimizing with covering indexes, 93

join nested loops, generated by, 239

key lookups, 53

nonclustered index seek + range scan + lookups, 81–90

RID lookups, 51

unordered nonclustered index scans + lookups, 91–93

loops

CTEs shortest-path solutions vs., 792–801

indexes for join nested loops, 239–240

iterative graph solutions with, 718, 721, 723–728, 730–733
moving average value computations

LOOP hint for nested loops join algorithm, 244–245
Nested Loops operators, 98, 183–186, 240–241
recursion vs., 718
shortest-path solutions vs. CTEs, 792–801
WHILE loops with TOP on key order, no key index, 581
lost updates, 645–646
LRU-K2 algorithm, 42

M
Machanic, Adam, 641
MAD (mean absolute deviation), 484, 523
many-to-one pivots, 304–307
Martin, Laurent, 456
materialized path model
adding leaves, stored procedure for, 744–745
advantages of, 743
ancestors, returning, 751–753
defined, 742
execution plans for, 750, 754
HIERARCHYID data type for implementing. See
HIERARCHYID data type
index creation for, 743–744
index key size limitations, 743
leaf nodes, returning, 751
level and path revisions, 745–748
performance issues, 752
queries, 749–754
shake effects, 743
sorting, 753–754
splitting paths, 752–753
subtrees, moving, 745–748
subtrees, removing, 748–749
subtrees, returning, 751
Max Degree of Parallelism setting, 175–178
MAX function
with CASE expressions, 301–302
window function (OVER) version of, 275–276
Max Worker Threads setting, 176–178
max_dop setting, 176
maximum concurrent interval problems, 456–465
MAXRECURSION hint, 728–729
mean absolute deviation (MAD), 484, 523
mean squared deviation (MSD), 484, 523
means
arithmetic, 479, 523
of continuous variables, 487–488, 509–512
MAD (mean absolute deviation), 484, 523
MSD (mean squared deviation), 484, 523
medians
continuous-distribution model, 368
defined, 479, 523
OFFSET-FETCH with APPLY solution, 370–371
PERCENTILE_CONT solution, 369
PERCENTILE_CONT solutions, 480–481
PERCENTILE_DISC function for, 480–481
ROW_NUMBER solution, 369–370
table for testing, 368–369
window functions for, 289–291
memory
leak handling by SQLOS, 586, 588
OLTP in. See In-Memory OLTP
memory optimized data filegroup, 675
MEMORY_OPTIMIZED option, 140
memory-optimized tables, parallelism inhibited by, 179
merge algorithm for joins, 241–245
MERGE statements
conflict prevention, 408–409
DELETE clauses with, 405–406, 414–416
full logging vs. other bulk methods, 376
INSERT clauses with, 405–406, 414–416
INTO updating persisted sets with, 336–337
MERGE INTO clauses, 405
MERGE ON predicates, 409–410
multiple source rows matched to target errors, 406
NULL handling, 407
ON predicates, 405
OPENROWSET BULK clauses with, 410–411
OUTPUT clause with, 414–416
SERIALIZABLE isolation level for, 409
sources of data allowed for, 410–411
UPDATE clauses with, 405–406, 414–416
USING clauses, 405, 410–411
WHEN MATCHED clauses, 405–407, 415–416
WHEN NOT MATCHED clauses, 404–407
MERGE TOP filters, 360
merging exchange variant of Gather Streams, 165–166
Microsoft Azure SQL Database alternative lock model, 636
Microsoft SQL Server. See SQL Server
Microsoft Virtual Academy (MVA), 224
Microsoft Visual Studio for CLR code, 589–591
Microsoft Windows time, 421–422
minimum missing values, finding, 195–199
missing values
finding, 195–200, 291–292
treatment of, 10
mixed extents, 43
modes
as centers of distributions, 479, 523
custom aggregate calculations of, 324–327
TOP WITH TIES to calculate, 480
MONTH function, 425
moving average value computations
advantages over other statistics, 512–513
defined, 523
exponential moving averages, 515–518
simple moving averages, 513–514
weighted moving averages, 514
window functions for, 274
MSD

MSD (mean squared deviation), 484, 523
multi-join queries
   DISTINCT clauses in, 231–232
   FORCE ORDER option, 236–237
   left outer joins followed by other joins, issues with, 234–235
   logical-join evaluation order, 234–237
   multiple ON clauses in, 236
   optimization, bushy plant, 236–237
   parentheses with, 235–236
   physical-join evaluation order, 232–233
   right outer joins for optimization, 235
multiple occurrences of rows
   allocation order scan sources of, 65–76
   index order scan sources of, 76–81
multistatement TVFs (table-valued functions), 550–553
multi-table queries
   APPLY operator based, 218–223
   joins in. See joins
   relational operators in. See relational operators
   subqueries as. See subqueries
table expressions as. See table expressions
multi-valued subqueries, 188–189
MVA (Microsoft Virtual Academy), 224

N

name resolution, implied, errors from, 201–202
National Institute of Standards and Technology (NIST), 707
natively compiled procedures environment,
   In-Memory OLTP
      advantages of, 690
      ATOMIC blocks, 698
cross-container queries, for, 703
cursors not supported, 704
performance vs. interop, 699
retry logic, 702
statistics generation, 697–698
stored procedures (SQL) not callable within, 705
TVPs for, 699–702
validation errors, 702
negative vs. positive logic, 200–201
nested loops join algorithms
   AND vs. BETWEEN in queries, 784
   forcing an optimization strategy, 244–245
   index performance issues, 239–241
Nested Loops operator
   bottlenecks, eliminating with Parallel APPLY Pattern, 183–186
   cardinality estimate problems with, 98
   parallel scans with, 169–170
nested sets graph solution
   advantages of, 778
   ancestors of nodes, returning, 786
dynamic trees defect, 778, 782
indexes for, 779
leaf nodes of roots, returning, 785
performance issues, 782–783
querying, 784–786
shake effect, 782
sort paths for nodes, 781
subordinates of nodes, counting, 786
subtrees of roots, returning, 784–785
value assignments, 778–784
nesting
   CTEs, 208
derived tables, issues from, 206
loops. See nested loops join algorithms
   of triggers, 577–578
   TRY-CATCH constructs, 664
.NET Base Class Library, 597
.NET Framework, Microsoft. See SQLCLR programming
Nevarez, Benjamin, 233
NEWID function, UDF prohibition of, 546
NEXT VALUE FOR function, 382–386
NIST (National Institute of Standards and Technology), 707
NO_BROWSETABLE option, 568
nodes
   adding, long paths resulting from, 767
   ancestor. See ancestors
   child. See child nodes
cross-container queries, for, 703
   connections between. See edges
defined, 707
descendant. See descendants
directed vs. undirected edges with, 707–708
   levels of, 718
   parent. See parent nodes
   subtrees, returning, 719–720
types of, 708
NOLOCK hints, 68, 72, 647–648
nonclustered index seek + range scan + lookups, 81–90
nonclustered indexes
   clustering keys, 52–53
covering nonclustered index seek + range scan, 96
data structure with, 43–44
   filtered, 120–122
   full scans of, avoiding, 153–156
   included non-key columns in indexes, 119–120
   In-Memory OLTP requirement for, 704
   key lists, restrictions on, 119
   key lookups, 53
   levels of, calculating, 49–50
   memory-optimized tables with, 677–680
   nonclustered index seek + range scan + lookups, 81–90
ordered covering nonclustered index scans, 63–65
row identifiers (RIDs), 51
seek operations in, 52–53
ordered covering nonclustered index scans

structured on B-trees, 52–53
structured on heaps, 50–51
uniquifiers, 52–53
unordered covering nonclustered index scans, 60–62
unordered nonclustered index scans + lookups, 91–93
non-equi joins, 230–231
NonParallelPlanReason node, 180–181
nonrepeatable reads, 645–646
normal distributions, 487–488, 509–512
NOT EXISTS clauses. See EXISTS predicate
NOT IN clauses, 203–204
NTILE OVER window function, 282–283, 285
null hypotheses, 495
NULLs
CONCAT_NULL_YIELDS_NULL set option, 568–570
ISNULL function, 323
IsNull property, SQLCLR, 596
missing value nature of, 10–11
NOT NULL constraints, 203–204
ORDER BY treatment of, 22
relational operators with, 249
subquery errors from, 203–204
UNION, EXCEPT, and INTERSECT operations on, 39
NUMERIC data type, 592–593
NUMERIC_ROUNDABORT option, 569
NVARCHAR type, 593–595

objects, page membership in, 42
OFFSET BY clauses, 3–5
offset window functions, 285–287
OFFSET-FETCH filters
2012 vs. 2014 versions of, 341
execution plans, 355–356
FETCH clause optional, 346
FIRST keyword, 345
index optimization, 357
isolation level effects on output, 354
median calculations with, 370–371
NEXT keyword, 345
offset required, 345
OffsetExpression property in plan, 354–357
ONLY keyword, 345
optimization of, 354–357
ORDER BY (SELECT NULL), 346
ORDER BY clause required, 345–346
ordering issues, 23–26, 345
paging solutions with, 345, 354–357
phase in logical processing, 6, 22–26
ROW/ROWS keywords, 345
similarity to TOP filters, 345
syntax for, 23, 345
ties not supported, 345
TO | AFTER imagined syntax, 357
Top operator to process, 354–357
OLAP (online analytical processing) Max Degree Of Parallelism setting, 178
OLTP workloads
In-Memory feature. See In-Memory OLTP
Max Degree Of Parallelism setting for, 178
merging data for, 404
ON predicate
inner joins, mandatory for, 228
missing values, treatment of, 10
multiple ON clauses in multi-join queries, 236
outer joins using, 230
phase in logical processing, 4, 9–13
UNKNOWN values, treatment of, 11
WHERE clauses, vs. in outer joins, 14–15
one-to-one pivots, 300–304
online transaction processing. See OLTP workloads
open schema, 300–301
OPENROWSET function
inserting data with, 378–380
MERGE statements using, 410–411
operators, query plan. See execution plans
optimistic concurrency model, 674
optimization of queries. See query tuning
OPTIMIZE FOR hint, 561–562
optimizer. See query optimizer
OPTIMIZER_WHATIF command (DBCC), 89–90
OPTION(LABEL = 'some label') hint, 556
OR disjunctions not allowed by In-Memory OLTP, 705
ORDER BY clause
with SQLCLR table-valued functions, 600
ORDER BY clauses
column aliases from SELECT phase, 17
cursors returned by, 21–22
DISTINCT clauses with, 20–22
dynamic sorting with, 542–546
grouping sets with or without, 337–338
logical order of, 3–5
logical phase order sample, 7
NULL value treatment, 11, 22
OFFSET-FETCH filter with, ordering issues, 22–26, 345–346
phase in logical processing, 6, 20–22
prohibited in set operation queries, 38–39
relational operators with, 249
SELECT @local_variable method with, 319–322
SELECT NULL trick, 285, 344
table expression inner query issues with, 204
TOP filter with, 22–26, 341–342
window functions in, 35–37
order of processing
logical order, 3–6
physical order, 2–3
typed order, 3
ordered clustered index scans, 62–63
ordered covering nonclustered index scans, 63–65
ordered set functions

ordered set functions, 290, 326–327
OUTER APPLY operator, 27–28, 221
outer joins
  FULL keyword with, 229
  LEFT keyword with, 229. See also left outer joins
logical processing phase, 13
not allowed by In-Memory OLTP, 705
ON clauses, 230
ON vs. WHERE clauses with, 14–15
physical-join evaluation order, 232–233
preserved tables, 13, 229–230
RIGHT keyword with, 229
syntax issues from, 224–225
type specification of, 13
WHERE clauses, 230
outer rows, 13
out-of-order pages, 47
OUTPUT clauses
  $action function in, 414–416
  chunking with, 414
  column identification in, 411
  DELETE statements with, 413–416
  flexibility of, 411
  INSERT clauses with, 412–416
  INSERT SELECT clauses with, 417
  INTO clauses with, 411, 412–413, 414
  MERGE statements with, 414–416
  UPDATE statements with, 411, 414–416

P

packing intervals for dates and times, 466–471
page free space (PFS), 45
pages
  elimination by In-Memory OLTP, 672–673
  extents of, 43
  out-of-order, 47
  splits, 47–48
  structure of, 42
  table structure with, 44–45
paging solutions
  with OFFSET-FETCH filters, 345
  optimization of OFFSET-FETCH for, 354–357
  optimization of ROW_NUMBER for, 358–360
  optimization of TOP for, 346–353
parallel query execution
  advantages of, 158
  Affinity Mask setting, 175, 177
  APPLY pattern, 181–186
  broadcast distribution scheme, 163
  Bulk Copy API, 173
  cardinality overestimations, effects of, 100
  cardinality underestimations, effects of, 99
  configuration guidelines, 177–178
  Cost Threshold for Parallelism setting, 175–176
  costs, parallel vs. serial executions, 87–90
  CPU costs, viewing, 176–177
  CXPacket data structure, 163
degree of parallelism (DOP), 159, 161–162, 164, 175
Demand distribution scheme, 163, 171–172
dependence on number of logical CPUs, 89–90
Distribute Streams variant of Exchange operator, 160–163
Exchange operators, 160–166
execution plan icons indicating, 159–160
factory line model of, 158
few outer rows optimization, 168–170
forward scans only rule, 116
Gather Streams variant of Exchange operator, 160–163, 165–166
hash distribution scheme, 163–165
Hash Join operators in, 170–171
Hash Match operator, 167
inhibitors of parallelism, 178–181
instance-level settings, 175–176
interval calculations with, 464–465
legacy vs. 2014 versions, 174
manual rewrites as alternative to query optimizer, 181–186
Max Degree of Parallelism setting, 175–178
Max Worker Threads setting, 176–178
max_dop setting, 176
memory-optimized table transactions, not used with, 696–697
merging exchange variant of Gather Streams, 165–166
nested loop solutions, 181–186
nonclustered index seek + range scan + lookups, 87–90
NonParallelPlanReason node, 180–181
optimization process, 176–177
packing tasks with, 470–471
Parallel APPLY Pattern, 181–186
Parallelism (Gather Streams) operator, 88, 160–163
partial aggregation, 166–167
partitioned tables, 171–173
–Pn startup parameter tool, 90
properties window features, 164
query cost metric, 178
query plans with, 159–160
range distribution scheme, 163
Repartition Streams operator, 161, 163, 169–170
Resource Governor settings, 175–176
round robin distribution scheme, 163, 169–170
row-distribution strategies, 163–165
scans, parallel, 168–170
SELECT INTO bulk operations, 173–174, 374
sorting, 165–166
stream-based model used by SQL Server, 159
threads per zone, 159, 161
trace flag 8649 tool, 90, 117
UDF inhibition of, 179–180, 548
XPacket data structure, 163
parameters
compiled values, 555–558
passed vs. declared within procedures, 559
preventing sniffing, 564–568
sniffing, 555–558
parent nodes
as ancestors. See ancestors
defined, 708
nested sets model of, 778
parent-child adjacency lists, converting to
HIERARCHYID, 771–773
PARSE function for dates and times, 431–433, 435
partial aggregation, 166–167
partitioning, ordering, covering pattern for indexing.
See POC (partitioning, ordering, covering) pattern for indexing
partitions
allocation unit types used by, 44
changes in, recompiles triggered by, 570–571
creating, 171–172
descending indexes with PARTITION BY, 117–118
parallel query execution with, 171–173
ranking with window functions, 281–285
structure of, 44
window functions with PARTITION BY, 263–266
paths. See also edges
ancestors, returning, 730–733
cycles in, determining if, 740–742
cyclic vs. acyclic, 708
HIERARCHYID data type for materialized path model. See HIERARCHYID data type
long paths issue, 767–771
materialized algorithms for solving. See materialized path model
multiple into a node, 711
subgraphs with paths, 733–736
topological sorts using, 736–739
transitive closure of directed. See transitive closure
Pearson chi-squared formula, 501–505
PERCENT input indicator, TOP filters, 342
percentages
absolute, 476–479
cumulative, 476–479
frequency calculations with PERCENT_RANK OVER, 477–478
grand totals, of, 268–269
PERCENT_RANK OVER window function, 288–289
percentiles
inter-quartile range calculations, 483
inverse distribution functions for calculating, 289–291
PERCENTILE_CONT for finding inter-quartile ranges, 483
PERCENTILE_CONT for finding medians, 369, 480–481
PERCENTILE_CONT window function, 289–291
PERCENTILE_DISC window function, 289–291, 480–481
performance
DMVs (dynamic management views), 134–139
query tuning for. See query tuning tools for measuring. See execution plans;
statistics, performance related
typical workloads, measuring. See Extended Events sessions
pessimistic concurrency model, 674
PFS (page free space), 45
phantom reads
defined, 645–646
Serializable level for preventing, 651–652
vs. isolation levels, table of, 646
phases, logical query processing. See logical query-processing phases
phone numbers, cleaning up, 549–550
physical execution plans, 97
physical layer, query tuning dependence on, 41
Physical Operation property, 239
physical processing order, 2–3
physical reads, 55
physical-query execution plans, 2
pivoting. See also unpivoting
actual frequencies, calculating with PIVOT operator, 503
CASE expressions, implicit use of, 30–31
custom aggregations with, 315–316, 318–319
dynamic, 530–535
FOR XML with, 318–319
implicit grouping phase, 30
many-to-one pivots, 304–307
one-to-one pivots, 300–304
phase in logical processing, 27, 29–31
PIVOT operator elements, 27
PIVOT operator syntax, 302–304
row counting with PIVOT, 534
spreading attributes, 301–302
use cases for, 299
Plan Explorer tool, 139
plan optimality, 568, 570–571
plan stability, 568–570
plans, parallel. See parallel query execution plans, query execution. See execution plans
POC (partitioning, ordering, covering) pattern for indexing
correlated subquery optimization, 192
ranking optimization with, 284–285
running totals optimization, 271–273
top N per group tasks, recommended for, 363–365
population moments, 479, 490. See also distributions
positive vs. negative logic, 200–201
Pötke, Marco, 456
power sets, CUBE function for creating, 331–333
PRECEDING keyword, 269–274, 277, 280–281
predicates, multiple, cardinality estimates for
predicates, multiple, cardinality estimates for, 104–107
prefetches
  joins generating, 241
statistics on, 55
preserved tables of outer joins, 13
primary keys
  clustered indexes resulting from, 43
  merges causing violations, 408–409
procedure statistics, 138
processing order, physical vs. logical, 2–3
processing queries. See logical query-processing phases
programmable objects
  Dynamic SQL. See Dynamic SQL
  SQLCLR. See SQLCLR programming
  stored procedures. See stored procedures
  triggers. See triggers
  user-defined functions. See UDFs (user-defined functions)
proleptic Gregorian calendar, use of, 419

Q
query cost metric, 178
query execution plans. See execution plans
query interop environment, In-Memory OLTP
  ATOMIC blocks, 698
  blocking, 691–693
  cardinality estimates, 697
  commit validation errors, 693–696
  cross-database query execution bar, 696
deadlocks, 691–693
  isolation levels available in, 691
  parallelism not used currently, 696–697
  similarity to standard SQL Server, 690
statistics generation, 697–698
query optimizer
  cardinality estimator component of, 97–98
cost breakdowns, 176
  Cost Threshold for Parallelism setting, 175–176
  flaws in, 154, 156
  manual rewrites as alternative, 181–186
  ordering issues, 543
parallel optimization process, 176–177
  parallel plans, criteria for, 175
plans created by. See execution plans
SQL Server, as component of, 2
  unnesting of subqueries, 156
query processing. See logical query-processing phases
query processor, query optimizer component of, 97
query tuning
  access method issues. See access methods
  allocation order scans, 65–76
bitmaps for, 170–171
cardinality estimates for. See cardinality estimates
clustered index seek + range scan, 93–94
  cost breakdowns, 176
covering nonclustered index seek + range scan, 94–97
CPU costs, viewing, 176–177
data structures. See internal data structures
date and time calculations, 445–446
DMVs (dynamic management views), 134–139
deadlocks, 691–693
early row reduction benefit, 166
efficiency for. See execution plans
extended events, prioritizing with, 131–134
Extended Events session statistics, 54, 56
filtered indexes, 120–122
execution plans for. See execution plans
hash join and bitmap in parallel queries, 170–171
histograms for. See histograms
included non-key columns in indexes, 119–120
index order scan safety issues, 76–81
inline index definition, 130–131
join order optimization, 232–233
multiplicity of tasks and optimal solutions, 368
nonclustered index seek + range scan + lookups, 81–90
Parallel APPLY Pattern, 183–186
parallel optimization process, 176–177
partial aggregation, 166–167
PARTITION BY clauses, 117–118
partitioned tables, parallelism for, 171–173
physical layer, importance of, 41
prioritization of queries with Extended Events sessions, 131–134
query cost metric, 178
query revisions for, 153–157
safety issues, allocation order scan, 65–76
SARGs, 445–446
scaling issue, 153
SELECT INTO bulk operations, 173–174
set-based vs. iterative solutions, 149–153
statistical vs. transactional query optimization, 473
statistics for. See statistics, performance related
storage engine scan treatment, 65–76
  stored procedure tuning, 554–568
strategies, testing for desired execution plans, 154–158
TOP filters to avoid unnesting, 156
unordered nonclustered index scans + lookups, 91–93
query_hash action, 131–134, 137–138
query-processing phases, logical. See logical query-processing phases
QUOTED_IDENTIFIER option, 569
QUOTENAME function, 318–319
R

RAISERROR command, 664–665
RAND function, UDF prohibition of, 546
range of a distribution, 482–483, 523
range parallel distribution scheme, 163
range predicate access method issues, 95–97
range scans. See scans
RANGE UNBOUNDED PRECEDING phrase, 279–280
RANGE unit for window frames, 276–280
rank, calculating
  CUME_DIST OVER, 288–289
  PERCENT_RANK OVER, 288–289
rank distribution window functions, 288–289
RANK OVER window function, 282–283, 285
window functions for, 281–285
RDBMS (relational database management system), 2
Read Committed isolation level, 646, 648–650, 660
Read Committed Snapshot isolation level, 646, 652, 655–656, 660
Read Uncommitted isolation level
  compared to other isolation levels, 646
  no shared locks, results of, 646–648
  NOLOCK resulting in, 68, 81
  shared locks not acquired on rows, 78–81
  unsafe category allocation order scans, 68–69
reads
  access method use of. See access methods
  DataAccessKind.Read, 179
dirty, 645–647
  isolation level settings for. See isolation levels
logical, 55, 57
phantom, 645–646, 651–652
read-ahead, 55
REBUILD keyword, 135
rebuilding indexes, 135
recompilations of stored procedures, 568–571
RECOMPILE option
  efficiency benefits from, 537–539
  issues with, 559
  statement-level vs. procedure-level, 537, 559
stored procedures, preventing plan reuse,
  558–560, 562–563
table variables with, 143–145, 148
unknowns, allowing sniffing of, 110
recovery models for database, 376–378
recovery models for transactions, 634
recursion
  ancestors, returning, 730–733
  limiting in an ON clause, 729
  MAXRECURSION hint, 728–729
  recursive CTEs, 209–211, 718–719, 722–723
  triggers, 577–578
reduce and conquer algorithms, 733
references
  column, table-qualified, 8
correlations, 189
regression, linear, 499–501
relational database management system (RDBMS), 2
relational division, 188
relational engine, SQL Server, 97
Relational Interval Tree (RI-tree) model, 456
relational model
  of missing values, 10
  SQL deviation from for duplicate rows, 18
relational operators
  EXCEPT, 249, 255–256
  input requirements, 249
  INTERSECT, 249, 252–254
  NULLs, treatment of, 11
  ORDER BY clauses with, 249
  precedence of, 249
  result column names, 249
  syntax of, 249
  UNION, 249–252
relations, application of operators to, 38
relative standard deviations, 486
REORGANIZE keyword, 135
Repartition Streams operator
  bottleneck from early use of, 183
  redistribution of streams role, 161, 169–170
  row-distribution strategies of optimizer, 163
too few rows triggering, 169–170
Repeatable Read isolation level, 646, 649–651, 691
REPLACE function, 597
Resource Governor parallelism settings, 175–176
RESULTS SETS clauses, 573–575
retry logic, 669–670
RIDs (row identifiers)
  nonclustered index seek + range scan + lookups, 81–90
  structure of, 51
right outer joins
  optimizing multi-join queries with, 235
  preserved tables, 13
  RIGHT keyword, 229
  vs. left outer joins, 230
RI-tree (Relational Interval Tree) model, 456
road system scenario, 715–718
rollbacks
  ROLLBACK TRAN command, 634–635
  sequence object issues from, 386
  temporary table and variable behavior, 148–149
ROLLUP function, 331–334, 337–338
root nodes, 708
root pages, 48
round robin parallel distribution scheme, 163, 169–170
rounding
  dates and times issues, 447–449
  ROUND function, 322
row constructor Standard SQL feature, 349
row identifiers (RIDs), 51, 81–90
row numbers

calculations of, 477–478
generation, determined vs. random, 283
window function for. See ROW_NUMBER OVER window function
@@rowcount, triggers using, 582
row-distribution parallel strategies, 163–165
ROW_NUMBER OVER window function
islands task solutions with, 292–294
maximum concurrent intervals task, 461–465
medians, calculating, 369–370
optimization of, 358–360
packing interval solutions, 466, 468–471
paging solutions with, 358–360
ranking function of, 282–285
separating elements with, 248
top N per group solution with, 364–365
topological sorts with, 736–739
row-offset arrays of pages, 42
ROW_OVERFLOW_DATA allocation units, 44
rows
errors due to access methods, 65–81. See also scans
estimated by histograms, 102–103
estimated from density vectors, 103
estimated in execution plans, 98
generating large numbers of with cross joins, 216–217
outer rows, 13
ranking. See ROW_NUMBER OVER window function
ROWS as a window frame unit, 269–276
ROWS UNBOUNDED PRECEDING phrase, 270–271, 273, 281
running total calculations
SUM OVER window elements example, 264
window functions for, 271–273, 477
year-to-date (YTD) calculation, 280–281

S

taxes

sample data generation with cross joins, 225
SARGs (search arguments), 445–446
SAVE TRAN command, 635
scalar subqueries, 187–188
scaling
databases, query tuning issue, 153–157
parallel stream-based model, 159
sort parallelism, 165–166
window functions vs. join and grouping for running totals, 273
scans
allocation order. See allocation order scans
backward, 116–117, 179
clustered index seek + range scan, 93–94
covering nonclustered index seek + range scan, 94–97
expensive part of plans, identification of, 154
fragmentation, 47–48
full, of nonclustered indexes, avoiding, 153–156
full scans of leaf level unclustered, expensiveness of, 154
full scans when no order required, 57–60
full table, resulting in deadlocks, 658
index order scans, 46–47, 59, 65, 76–81
ordered clustered index scans, 62–63
ordered covering nonclustered index scans, 63–65
parallel, 168–170
RIDs, using in, 51
scan count statistic, 55
storage engine treatment of, 65–81
table scan/unordered clustered index scan access method, 57–60
unordered covering nonclustered index scans, 60–62
unordered nonclustered index scans + lookups, 91–93
SCHEMABINDING option, views, 212
scope
batches, relationship to, 525
SCOPE_IDENTITY function, 412
temporary objects (tables and variables), 140, 148
scans
allocation order. See allocation order scans
backward, 116–117, 179
clustered index seek + range scan, 93–94
covering nonclustered index seek + range scan, 94–97
expensive part of plans, identification of, 154
fragmentation, 47–48
full, of nonclustered indexes, avoiding, 153–156
full scans of leaf level unclustered, expensiveness of, 154
full scans when no order required, 57–60
full table, resulting in deadlocks, 658
index order scans, 46–47, 59, 65, 76–81
ordered clustered index scans, 62–63
ordered covering nonclustered index scans, 63–65
parallel, 168–170
RIDs, using in, 51
scan count statistic, 55
storage engine treatment of, 65–81
table scan/unordered clustered index scan access method, 57–60
unordered covering nonclustered index scans, 60–62
unordered nonclustered index scans + lookups, 91–93
SCHEMABINDING option, views, 212
scope
batches, relationship to, 525
SCOPE_IDENTITY function, 412
temporary objects (tables and variables), 140, 148
search arguments, 445–446
second regression line, 500
security issues
Dynamic SQL EXEC parameters, 526–528
EXECUTE AS clauses, 546
SQLCLR stored procedure credentials, 614
seeks
clustered index seek + range scan access method, 93–94
cost in an index, 154
cost in reads, 49–50, 82
covering nonclustered index seek + range scan access method, 94–97
forcing as part of a strategy, 154–155
generated by POC index strategies, 364
index seek access method, 49
looking for particular key in nonclustered indexes, 52
operators using. See Index Seek operator
Segment operator, 64
Seidl, Thomas, 456
SELECT clauses
all-at-once operations, 17–18
AS with, 17
column aliases and processing order issue, 14
column references with, 17
Evaluate Expressions phase, 17–18
evaluate expressions phase, 6
INTO with. See SELECT INTO command
logical order of, 3–5
ORDER BY clauses with, 20–22
spooling optimization for window functions

phase in logical processing, 4–5, 17–20
sample query for logical phases, 7
SELECT @local_variable method, 319–322
window functions in, 35–37

SELECT INTO command
bulk operations, 173–174
deduplication with, 400–401
drawbacks of, 375
eager writes, 140, 374–375
execution plans, 374
IDENTITY function with, 381
identity property copying, 374
In-Memory OLTP, not supported by, 705
INSERT SELECT as alternative, 374–375
logging, measuring, 377–378
logging options, 374
materials copied by, 373
metadata access blocking issue, 375–376
parallelism capability of, 374
self joins, 230
self-contained subqueries, 187–189
semi joins, 237–238
separating elements of arrays tables, 245–249
SEQUEL (Structured English QUEry Language), 2–3
sequence objects
ALTER SEQUENCE command, 382, 384
aspects and properties of, 394–395
cache performance issues, 387–394
CREATE SEQUENCE command, 382
CYCLE option, 383
data types supported by, 382
default values, 382
identity property, compared to, 384, 390–395
INCREMENT BY property, 383
keys, automating creation of, 384–385
MAXVALUE property, 382–383
MINVALUE property, 382–383
missing values, causes of, 387
NEXT VALUE FOR function, 382–386
NULLs allowed in target columns, 385
overwriting existing keys with, 385
performance considerations, 387–394
rolled back transaction issues, 386
START WITH property, 383
storing values for use, 385
syntax for creating, 382
sys.Sequences view metadata, 384
sequences of numbers, generating
identify property for. See identity property
performance considerations, 387–394
row numbers. See row numbers
sequence objects for. See sequence objects
table expressions for, 215–218
Serializable isolation level
comparing to other levels, 646
implementing, 651–652
merger conflict prevention with, 409
query interop environment with, 691
session contexts, 584
set-based approaches
nested sets graph solution, 778–786
separating elements problem, 245–249
sets defined, 149
vs. iterative solutions, 149–153
SGAMs (shared global allocation maps), 42
Shannon, Claude E., 518
shared global allocation maps (SGAMs), 42
shared locks (S), 636–639, 648–649
shared with intent exclusive locks (SIX), 637
shortest-path solutions, 791–801
Showplan Statistics Profile events, 138
Showplan XML Statistics events, 138
SHOW_STATISTICS command, 101–102
sibling nodes
defined, 708
sorting within, 736–739
significance, statistical, 498, 502, 509–512
Simple data recovery model
cache performance test with, 390–394
SELECT INTO operations with, 378
simple moving averages (SMAs), 513–514
SIMPLE recovery model for transactions, 634
SINGLE_ options, OPENROWSET function, 379–380
skewness, 479, 488–489, 490–495, 523
skipping rows
allocation order scan sources of, 65–76
index order scan sources of, 76–81
slopes, 499–501
SMALLDATETIME data type, 419–420
SMALLDATETIMEFROMPARTS function, 431
SMAs (simple moving averages), 513–514. See also
moving average value computations
Snapshot isolation level, 646, 652–655, 691
sniffing, disabling, 110
Sommarskog, Erland, 249, 542, 573
sorting
cardinality overestimations, effects of, 100
cardinality underestimations, effects of, 99
carry-along sorts, 366–368
dynamic, 542–546
graphs, hierarchical, 736–739
IP addresses, 773–777
nonlinear scaling of Sort operation, 183
ORDER BY for. See ORDER BY clauses
parallel execution of, 165–166
separated lists of values, 773–777
Sort operator, expense of, 326–327
TOP filter ordering issues, 22–26
sp_executesql procedures, 529–530, 539–540
splits, page
allocation order scan errors from, 65–68
causes of, 47–48
spooling optimization for window functions,
spreading attributes, spreading attributes, 301–303, 305, 315–316
spreads of distributions
  inter-quartile ranges, 483
  mean absolute deviations, 484
  mean squared deviations, 484–485
  ranges, 482–483
  standard deviations, 486–487, 490, 498, 511, 523
  variances, 485–486, 505–508
sp_statement_completed events, 131
SQL, 1–2. See also T-SQL (Transact SQL)
SQL Database
  alternative lock model, 636
  default isolation level, 646
SQL injection attacks, 526–530, 533
SQL Server
  Data Tools (SSDT), 589
  Management Studio. See SSMS (SQL Server Management Studio)
  query optimization component of. See query optimizer
  relationship to T-SQL, 1
SQL injection attacks, 526–530, 533
SQL Sentry, 139
SQL CLR programming
  advantages of scalar functions, 597
  advantages over UDFs, 550
  AppDomain management and scope, 587, 589
  architecture of, 585–588
  assembly hosting, 587–588
  AUTHORIZATION options, 588
  calling CLR functions, 592, 600
  CAS (Code Access Security), 588
  case sensitivity issues, 595–597
  context connection strings, 603–605, 607
  CREATE ASSEMBLY command, 587–588, 591
  CREATE FUNCTION statements, 590–591
  creating table-valued functions, 598–600
  database access by functions, 603–605
  DataRow type, 604
  date and time types issues, 595
  DLLs in, 587, 591
  enabling code, 587
  EXTERNAL_ACCESS permission set, 587–588, 605
  fill row method, 598–599, 604
  FillRowMethodName option, 599
  function creation, 589–590
  HostProtection attribute, 588
  IEnumeratable values, 598
  indexes on functions, setting, 590
  in-process hosting model, 586
  IsNull property, 596, 599
  memory utilization, 600–603
  modifying of databases capability. See SQLCLR stored procedures
  .NET Base Class Library, 597
  .NET basis of, 585–586
  nullable types, 595
  NUMERIC type issues, 592–593
  offloading to an application server, 597
  ORDER BY clause with, 600
  ownership of assemblies, 588
  PERMISSION_SET options, 587–588
  project properties, setting, 591
  publishing assemblies, 591–592
  query plans of scalar functions, 588–589
  read-only nature of access, 604
  relation to T-SQL, 585
  remote server access, 605
  SAFE permission set, 587, 591
  sandboxes, memory, 588
  scalar function use cases, 597
  security privileges of, 586
  SqlCompareOptions enumeration, 595–597
  SqlFacet attribute, 593, 596
  SqlFunction attribute, 590–595, 599
  SQLOS integration, 586
  SqlString type, 590, 595–597
  SqlTypes namespace, 590–595
  SSDT projects, 589–590
  stored procedures. See SQLCLR stored procedures
  string splitting, 598–603
  string type issues, 593–595, 595–597
  stub scripts, custom, 595
  TableDefinition, 599
  table-valued functions, streaming, 598
  TRUSTWORTHY mode, 605
  T-SQL stub function creation, 591–592
  type compatibility with SQL Server, 590–595
  type fixes, making in Visual Studio, 593
  UDF CLR scalar functions, 588–596
  UNSAFE permission set, 587–588
  user-defined aggregates. See SQLCLR user-defined aggregates
  user-defined types. See SQLCLR user-defined types
  Value property of SqlTypes, 594
  SQLCLR stored procedures
  comparing to other options, 605
  context connections, 607
  creating, 606–607
  @@ERROR function, 613
  EventData property, 616
  exception handling, 609–613
  ExecuteAndSend method, 610–612
  executing, 606–607
  impersonation, 614
  modifying of databases capability, 605
  publication of, 606
  RAISERROR issues, 611–612
  return values, 606
  security credentials with, 614
  Send methods, 607–609
  SqlContext class, 608, 614–616
  SqlPipe class, 607–609
  templates for, 606
  THROW statements, 613
statistics for BI

triggers, 615–616
try-catch blocks for exceptions, 612–613
SQLCLR user-defined aggregates
Accumulate method, 629–630
binary formatting, 628–629
collection classes in .NET for strings, 629–630
concatenation with comma separation, 629–632
creating, 628–629
duplicate value options, 631
IBinarySerialize interface implementation, 630–631
Init method, 628, 630, 632
InvariantToOrder option, 631
life cycles of, 628–629
Merge method, 629–630
methods, mandatory, 628–629
query plan limitation, 631
Read method, 631–632
schema reference requirement, 629
serialization requirement, 629
SqlString default type, 629–630
SqlUserDefinedAggregate settings, 631
template for, 628
terminate method, 629–630
testing, 632
Write method, 631–632
SQLCLR user-defined types
byte ordered option, 625
case sensitivity of method and property names, 621
complex type issues, 621–627
conversion issues, 620
creating, 617–618
deploying, 619–620
factory methods, 624–625
functionality of, 617
IBinarySerialize interface, 625–626
instantiating, 620
INullable interface, 618
Native formatting, 620–621
Null property, 618
Parse method, 618–619, 621, 623–624
public accessor properties, 623–624
publishing, 619–620
Read method, 626–627
required interfaces, attributes and methods, 618
schema of, 620
Serializable attribute, 618
SqlUserDefinedType attribute, 618, 625
street address complex types, 621–625
stubs for, 620
template for, 617–618
ToString method, 618–619, 621, 627
UserDefined byte size, 625
Write method, 626–627
SqlFacet attribute, SQLCLR, 593, 596
SQLOS (SQL operating system), 586, 588
SQLOS schedulers, 175–177
sql_statement_completed events, 131
SqlString type, SQLCLR, 590, 595–597
SqlTypes namespace, SQLCLR, 590–595
SQL_VARIANT type, 542–543
SSDT (SQL Server Data Tools), 589
SSMS (SQL Server Management Studio)
  Display Estimated Execution Plan, 53
  Include Actual Execution Plan option, 138–139, 151–152
  performance of queries, measuring, 54
  standard deviations, 486–487, 490, 498, 511, 523
  standard normal distributions, 487–488
  standard SQL, 1
  star join queries
    bitmaps for parallel optimization, 171
    columnstore performance statistics, 130
    rowstore vs. columnstore efficiencies, 123–127
  statement completed events, Extended Events
    sessions, 131–134
statistics for BI (business intelligence)
  analysis of variance, 505–508, 523
  cases, 473–474
  centers of distribution, 479–481
  chi-squared tests, 501–505, 523
  coefficient of determination (CD), 498–499, 523
  coefficient of the variation (CV), 486, 523
  contingency tables, 501–505, 523
  correlation coefficients, 498–499, 523
  covariance, 495–498, 523
  CUME_DIST frequency calculations, 477–478
  definite integration, 509–512, 523
  degrees of freedom, 485, 502, 505–508, 523
descriptive statistics for continuous variables, 479–494
  discrete variables, 474
  entropy calculations, 518–521, 523
  frequency calculations, 476–479, 523
  frequency distributions, 477–479
  F-tests, 506–508, 523
  higher population moments, 487–494
  histograms, 476–479
  importance of, 473
  inter-quartile ranges, 483, 523
  kurtosis, 489–495, 523
  linear dependencies, 495–511
  linear regression, 499–501, 523
  mean absolute deviations, 484, 523
  mean squared deviations, 484–485, 523
  means. See means
  medians. See medians
  modes, 324–327, 479–480, 573
  moving averages. See moving average value computations
  normal distributions, 509–512
  PERCENT_RANK frequency calculations, 477–478
  preparing data for, 473–474
  ranges of distributions, 482–483, 523
statistics, performance related

- statistics for BI (business intelligence), continued
  - row number calculations, 477–478
  - running totals, 477
  - SalesAnalysis view sample data, 474–475
- significance, statistical, 498, 502, 509–512
- skewness, 479, 488–489, 490–495, 523
- slopes, 499–501
- spread of distribution, 482–486
- standard deviations, 486–487, 490, 498, 511, 523
- variables, 473–474
- variances, 485–486, 505–508, 523
- window functions for data calculations, 288–291

- ascending key problems, 107–110
- AUTO_CREATE_STATISTICS option, 102, 111
- cardinality. See cardinality estimates
- density vectors, 102, 103
- dm_db_index_operational_stats function, 135
- dm_db_index_physical_stats function, 135
- dm_db_index_usage_stats function, 135–136
- dm_db_missing_index_columns function, 136–137
- dm_db_missing_index_details view, 136
- dm_db_missing_index_group_stats view, 136
- dm_exec_procedure_stats, 138
- dm_exec_query_profiles view, 138–139
- dm_exec_query_stats view, 137–138
- dm_exec_trigger_stats, 138
- DMFs (dynamic management functions), 134–139
- DMVs (dynamic management views), 134–139
- DROP command, 111
- filtered, creating, 120
- filtered index, 120–122
- headers, 102
- histogram. See histograms
- indexes, creation with, 101
- main types created by SQL Server, 101
- output interpretation, STATISTICS option, 55
- refresh rate issues, 107–110
- SHOW_STATISTICS command, 101–102
- sniffing, disabling, 110
- STATISTICS IO option, 55–56
- STATISTICS IO spooling optimization stats, 279
- STATISTICS PROFILE option, 138
- STATISTICS TIME option, 55–56
- STATISTICS tool syntax, 54
- STATISTICS XML option, 138
- for temporary objects (tables and variables), 143
- trace flag 2371, 110
- trace flag 2389, 109
- unknowns in cardinality estimates, 110–115
- variables as unknown values, 104, 110
- WHERE clause predicate, 120
- STDEV function, 486–487
- STDEVP function, 486–487
- step numbers, logical query-processing phases, 3
- storage engine
  - clustered index seek + range scan, 93–94
- consistency requirements, 65
- covering nonclustered index seek + range scan, 96
- NOLOCK effect, 68, 72
- Read Uncommitted isolation level, 68–69, 78, 81, 647–648
- safe category, 69–70, 72
- TABLOCK effect, 69
- treatment of scans, 65–81
- unordered nonclustered index scans + lookups, 91–92
- unsafe category, 68, 72–74
- stored procedures
  - adding leaves to graphs, 744–745
  - advantages of, 553
  - cardinality estimates, implications of inaccuracy, 557–558
  - covering indexes for, 558
  - database designation for execution, 533
  - deployments of changes to, 553
  - dynamic filtering example, 535–542
  - dynamic pivoting example, 531–533
  - dynamic sorting example, 542–546
  - execution plan reuse, 554–558
  - initial compilations of, 554
  - iterative graph solutions with, 718–719
  - KEFPFIXED PLAN hint, 570–571
  - native compilation by In-Memory OLTP, 673
  - not supported within In-Memory OLTP natively
  - compiled processes, 705
  - parameter sniffing, 555–558
  - parameter sniffing, preventing, 564–568
  - parameterized queries in, 553
  - parameters passed vs. declared within
  - procedures, 559
  - preventing execution plan reuse, 558–560
  - recompilations, 568–571
  - RECOMPILE option, 558–560, 562–563
  - set options, plan affecting, 568–569
  - sp_statement_completed events, 131
  - SQLCLR. See SQLCLR stored procedures
  - transactions using, 635–636
  - triggered by events. See triggers
  - tuning, 554–568
  - TVPs in, 571–573
  - variable sniffing, lack of, 560–564
- Stream Aggregate operator, 153, 167, 272
- streaming table-valued functions, SQLCLR, 597–605
- street addresses, 621–622
- string concatenation
  - COALESCE function, 316
  - CONCAT function, 316
  - concatenation with comma separation, 629–632
  - cursors for, 314–315
  - FOR XML method, 317–319
  - pivoting for, 315–316, 318–319
  - SELECT @local_variable method, 319–322
  - separators, adding, 318
types of custom aggregate calculations of, 313–314
strings
  cleaning, 549–550
collection classes in .NET for, 629–630
comma separated values SQLCLR examples, 598–603
REPLACE function, String.Replace SQLCLR function performance vs., 597
separating elements of, 245–249
SQLCLR string type issues, 593–597
Structured English QUEry Language (SEQUEL), 2–3
structures, internal data. See internal data structures
STUFF function, 318, 549–550
subqueries
  correlated, 187, 189–194
cross joins for optimization, 226–227
EXISTS predicate with, 194–201
identifying gaps problem, 198–200
In-Memory OLTP, not supported by, 705
multi-valued, 188–189
nesting capabilities of, 187
NULLs, errors from, 203–204
optimizing Index Seek operators, 192–193
POC (partitioning, ordering, covering) pattern for indexing, 192
scalar, 187–188
scalar aggregate, 191
self-contained, 187–189
substitution errors in column names, 201–202
table expressions with, 204–205
TOP filter based, 191–194
troubleshooting self-contained vs. correlated, 187, 189
SUBSTRING function, 246–248
subtrees. See also trees
  moving, 745–748
  removing, 748–749
  returning, 719–720
SUM function, 261–262, 265–268
SUM OVER window function
  advantages over SUM function, 263–264
  frame delimiters with, 269–272
  GROUP BY with, 267–269
  islands solution with, 296–299
  maximum concurrent intervals task, 459–461
  packing interval solutions with, 466–468
  PARTITION BY with, 263–265
  performance of, 266–268
  window elements supported by, 264
surface-area restrictions, In-Memory OLTP, 703–705
swapping column values, 18
SWITCHOFFSET function, 425–426
sys.allocation_units view, 42
SYSDATETIME function, 422
SYSDATETIMEOFFSET function, 421–424
sys.partitions view, 42
sys.system_internals_allocation_units view, 44
SYSUTCDATETIME function, 422

T

table expressions
  CTEs. See CTEs (common table expressions)
  derived tables. See derived tables
  inline TVFs. See inline TVFs (table-valued functions)
  kinds of, 204
  query requirements for, 204–205
  updating data with, 402–403
  views. See views

table operators
  Cartesian product logical processing phase, 8–9
  evaluation order of, 27
  joins. See joins
  logical query processing of, 8–14
  logical query-processing phase for, 26–35

table scans
  Table Scan operators, 65
  table scan/unordered clustered index scan access method, 57–60

table variables
  advantages over temporary tables, 143
cardinality estimates, 143–146
declaring, 144
declaring with table types, 571–572
  inline index definition with, 130–131
  INTO clauses directing rows to, 412–413
  parallelism inhibited by modification of, 179
  RECOMPILE option, 143–145
  rollbacks with, 148–149
  scopes of, 140, 148
  statistics available for, 143
  table types for, 571–573
  temporary. See temporary table and variable objects
  trace flag 2453, 146
  transactions failures, advantages with, 148–149
  as TVF outputs, 550–553

tables
  allocation units, 44
  B-trees, 43–44, 46–50
  changes in, recompiles triggered by, 570–571
  CTEs. See CTEs (common table expressions)
  deleting all rows from with TRUNCATE TABLE, 395–399
  expressions. See table expressions
  heap data structure, 43–46
tables, continued

HOBT (heap or B-tree) organization, 43–44
IAMs (index allocation maps), 44–45, 47
inline TVFs. See inline TVFs (table-valued functions)
internal data structures for, 43–53
memory optimized. See In-Memory OLTP
memory optimized, defining as, 140
memory optimized, parallelism inhibited by, 179
MEMORY_OPTIMIZED option, 572
operators. See table operators
partitioned. See partitions
scans. See table scans
temporary. See temporary table and variable objects
temporary local, with triggers, 584–585
TVPs (table-valued parameters), 571–573
type definitions, 571–573
variables. See table variables
views. See views
table-valued functions, SQLCLR, 597–605
Table-valued parameters (TVPs), 571–573
TABLOCK hint, 69
tempdb
• batch spills to, 128–129
cardinality estimate effects, 99–100
tables in not indexed, issues from, 581
temporary object use of, 140
temporary table and variable objects
caching mechanism issues, 147–148
cardinality estimates with, 139
cases to use for query tuning, 139
CTE expensive work, avoiding repetition of, 140–143
disk activity from, 140
eager writes, 140
global temporary tables, 148
local temporary table qualities, 139–140
non-tuning usage of, 139
RECOMPILE option, 143–145, 148
rollbacks with, 148–149
scope of, 140, 148
statistics maintained for, 143
table cardinality estimates, 146–147
table execution plans, 147
table expression qualities, 139–140
table variable qualities, 139–140
tables vs. variables, performance, 143
tempdb use, 140
types of temporary objects, 139–140
threads
• CXPacket data structure, 163
distributor or coordinator threads, 164–165
Max Degree of Parallelism setting, 175–178
Max Worker Threads setting, 176–178
parallel query execution number of, 159, 161, 164
quantum punishment of, 586
three-value logic, 10

THROW statements, 613, 665
tile numbers, NTILE for, 282–283
TIME data type, 419–420, 422, 440, 595
TIMESTAMP PART function, 431
times. See dates and times
TODATETIMEOFFSET function, 426
TOP filters
• anchor-based paging strategy, 346–352
• chunked modification with, 361–363
• DELETE statements with, 360–363
• DESC option, 344–345
elements of, 22
• execution plan for TOP over TOP, 353
• execution plans for anchor sorts, 348, 351, 352
• index choices, effects on optimization, 349
• inner ORDER BY clauses do not guarantee order, 344–345
• INSERT TOP statements, 360
• MERGE TOP statements, 360
• minimum missing values, finding with, 196–198
• modes, TOP WITH TIES to calculate, 480
• modification statements of, 360–363
• nested TOP paging strategy, 352–353
• optimization of paging solutions, 346–353
• ORDER BY clauses with, 341–342
• ORDER BY (SELECT NULL), 344
• ordering issues, 22–26, 342–343
• parallelism inhibited by certain, 179
• PERCENT input indicator, 342
• phase in logical processing, 6, 22–26
• predicate phrasing, effects on performance, 349–352
• proprietary to T-SQL, 341
• suggestion to add OVER to, 366
• syntax of, 341
• tie breaking with subqueries, 191–194
• Top Expression property of Top operator, 348
• Top operator in execution plans, 64, 348
• unnesting of subqueries, avoiding with, 156
• UPDATE TOP statements, 360
• WHILE loops with TOP on key order, no key index, 581
• WITH TIES option, 343–344
top N per group tasks
• APPLY with TOP solutions, 365–366
carry-along sorts for, 366–368
• concatenation solutions, 366–368
• CROSS APPLY solution to, 219–220
density of partitioning elements, 363–364
• elements of, 363
• factors influencing efficiency of T-SQL solutions to, 363–364
• N=1 solution, 366–368
• POC index support for, 192, 363
• ROW_NUMBER based solution, 364–365
topological sorts, 736–739
• trace flag 272, 382, 387
• trace flag 237, 110
trace flag 2389, 109
trace flag 2453, 146
trace flag 8649, 90, 117, 465
Transact SQL (T-SQL). See T-SQL (Transact SQL)
transactional replication bulk-logging requirements, 376
transactions
ACID properties, list of, 633
AFTER DML triggers with, 575–578
atomicity property, 633
autocommit default mode, 633
BEGIN TRAN command, 633–636
BULK_LOGGED recovery model for, 634
COMMIT TRAN command, 634–636
commit validation errors, 693–696, 702
completion guarantees, 634
consistency property, 634
constraints, immediacy of, 634
deadlocks of. See deadlocks
delayed durability, 634
doomed state, 666
durability property, 634
deadlocks of. See deadlocks
defined, 633
doomed state, 666
duration property, 634
doomed state, 666
failed state, 666
FULL recovery model for, 634
IMPLICIT_TRANSACTIONS option, 633
In-Memory OLTP of. See In-Memory OLTP
isolation level settings. See isolation levels
isolation property, 634
locks from. See locks
log buffers, flushing of, 634
nesting not supported, 634–635
recovery models, 633–634
redo phase of recovery process, 633
ROLLBACK TRAN command, 634–635
SAVE TRAN command, 635
savepoints, 635–636
SIMPLE recovery model for, 634
stored procedures in, 635–636
@@trancount, 635, 666–667
undo phase of recovery process, 633
UPDATE variable method for gapless sequences, 403–404
XACT_ABORT option, 666–667
XACT_STATE option, 667–668
transitive closure
cycle-detection logic for, 793
defined, 787
directed acyclic graphs, of, 787–792
distance calculations, 789–790
duplicate edge elimination, 789
efficiency of solutions, 787
filtering shortest paths, 791–792
materializing result sets, 799
shortest-path solutions, 791–801
undirected cyclic graphs, 792–801
trapezoidal rule, 509–510
trees
B-tree table representations. See B-trees
components of, 708
defined, 708
employee organizational chart scenario, 709–711
HIERARCHYID for representing. See
HIERARCHYID data type
materialized path solutions for. See materialized path model
nested sets graph solution, 778–786
parent-child adjacency lists, converting to
HIERARCHYID, 771–773
undirected cyclic graphs, 792–801
troubleshooting of SQL Server, 559–563
transaction errors in TRY blocks, 666–668
trapped errors not reported, 664
UDF prohibition of, 546
triggers
AFTER DDL triggers, 579–581
AFTER DML triggers, 575–578, 581–585
AFTER UPDATE triggers, 576–577
CREATE TRIGGER command, 576–578
cursors with, 581–583
defined, 575
disabling, 583–584
INSTEAD OF DML triggers, 578
integrity enforcement for, 583
nesting and recursion of, 577–578
ON ALL SERVER, 580
ON DATABASE, 580
performance issues for, 581–585
rollbacks with, 149
session contexts, 584
SQLCLR based, 615–616
temporary local tables with, 584–585
TRUE value, 10
TRUNCATE TABLE statements
DML statement nature of, 399
identity property handling, 396–397
keys and views preventing execution, 397–398
partition switching alternative to, 397–399
permissions required for, 395
speed advantage over DELETE statements, 395
syntax of, 395
TRY_CAST function, 432
TRY-CATCH constructs
advantages over @@error function, 662–663
ERROR_ functions with, 664
ERROR_NUMBER functions in, 664–666
mechanics of, 663
nesting, 664
non-trappable errors, 664
retry logic, 669–670
syntax, 663–664
THROW statements in, 665–666
transaction errors in TRY blocks, 666–668
trapped errors not reported, 664
UDF prohibition of, 546
TRY_CONVERT function, 431–432
T-SQL (Transact SQL)
  inefficiency of iterative solutions, 149–150, 152–153
  relation to industry standards, 1
  tuning queries. See query tuning
tuples, 18. See also rows
TVFs (table valued functions), 550–553
TVPs (table-valued parameters)
  for In-Memory OLTP procedures, 699–702
  for stored procedures, 571–573
type-conversion errors, 542
typed order vs. logical order, 3

U
UDAs (user-defined aggregates) for skewness and kurtosis, 490–495
UDFs (user-defined functions)
capabilities of, 546
  CLR-based. See SQLCLR programming
  inline, encapsulating, 719
  iterative graph solutions using, 718–719
  iterative logic in, 549–550
  limitations of, 546–547
  multiple statements in, 549–550
  parallelism inhibited by, 179–180, 548
  performance penalties associated with, 547–549
  scalar-valued, 546–550
  subgraph solution, 718–724
  TDFs as alternatives to, 548–549
UNBOUNDED keyword, 269–272, 277–281, 286
undirected cyclic graphs
  conversion to directed graphs, 792–793
  road system scenario, 715–718
  transitive closure of, 792–801
  weighted, 718
undirected graphs, 708. See also undirected cyclic graphs
uniform extents, 43
UNION operator
  ALL variant, 37–38, 250
  CHECK constraints with, 251–252
  costs, 251
  DISTINCT implied, 250
  execution plans for, 251–252
  grouping sets with, 329–330
  logical query-processing phase for, 38–39
  NULLs, treatment of, 11, 249
  ORDER BY with, 249
  precedence, 249
unions not allowed by In-Memory OLTP, 705
UNIQUE constraints
  filtered indexes with, 122
  not supported by In-Memory OLTP, 704
  NULL value treatment, 11
  uniquifier columns, 46–47, 52–53
UNKNOWN value, 10–11
unknowns in cardinality estimates, 110–115
unordered clustered index scans, 57–60
unordered covering nonclustered index scans, 60–62
unordered nonclustered index scans + lookups, 91–93
unpivoting. See also pivoting
  with CROSS APPLY and VALUES, 310–311
  with CROSS JOIN and VALUES, 308–310
  purpose of, 307–308
  with UNPIVOT operator, 312–313
  UNPIVOT operator elements, 27
  UNPIVOT operator phase in logical processing, 31–35
unrestricted cross joins. See Cartesian products
UPDATE statements
  AFTER UPDATE triggers with, 575–578
  INSTEAD OF UPDATE triggers with, 578
  in MERGE statements, 405–406
  with OPENROWSET function, 380
  OUTPUT clauses with, 411
  overwriting existing keys with sequence objects, 385
  swapping column values, 18
  UPDATE TOP filters, 360
  updating tables with CTEs, 402–403
  variable assignment method, 403–404
updating data
  Repeatable Read isolation level for preventing conflicts, 650
  Snapshot isolation level for preventing conflicts, 652–655
  as a source of row return errors, 76–81
  table expressions for, 402–403
  UPDATE function test for updated columns, 577
  update locks (U), 637
  variable assignment method for, 403–404
  UPDLOCK hints, 650–651, 757–758
user-defined functions. See UDFs (user-defined functions)
user-defined types, SQLCLR. See SQLCLR user-defined types
USING clauses, 410–411
UTC (Coordinated Universal Time), 420–421

V
validation errors, 693–696, 702
.value method, 317–318
variables
  chi-squared tests of independence of, 501–505
  as columns, 473
  continuous, 474
  correlations between, 495–501
  discrete, 474
  independence of, assumptions about, 499
  relationships between. See linear dependencies
window functions

sniffing, lack of, 560–564
temporary table. See temporary table and variable objects
variances
analysis of, 505–508
covariance, 495–498
defined, 523
VAR and VARP functions, 485–486
vector expression Standard SQL feature, 349
vertexes. See nodes
views
CHECK option, 212
data not stored, just metadata, 212
input parameter support lacking, 215
INSTEAD OF DML triggers, 578
of joined partitioned tables, 212–214
performance issues, 213–214
query requirements for, 204–205
reusable nature of, 211–212
SCHEMABINDING option, 212
scopes of, 204
virtual tables
add outer rows results, 13
APPLY operator generation of, 26–29
FROM clauses, generated by, 8–14
generation, 4–6
GROUP BY phase, 15–16
PIVOT operator generation of, 29–31
TOP filters, generated by, 22–26
UNPIVOT operator generation of, 31–35
WHERE phase, 14–15
Visual Studio for CLR code, 589–591

W

Waymire, Richard, 528
weekdays, calculating, 423, 436–439, 441–445
weeks, grouping by, 449–450
weighted graphs, 714, 718
weighted moving averages (WMAs), 514
WHEN NOT MATCHED BY SOURCE clauses, 404–407
WHERE filters
column IS NOT NULL, 122
dates and times filtered by, 445–446
groups not allowed in, 14
inner joins with, 228
logical order of, 3–5
logical phase order sample, 7
NULLs, subquery errors from, 203–204
ON clause, vs. in outer joins, 14–15
outer joins with, 230
phase in logical processing, 4–5, 14–15
UNKNOWN values, treatment of, 11
window functions using, 263–264
WHILE loops with TOP on key order, no key index, 581
White, Paul, 148, 349, 543

window aggregate functions. See aggregate window functions
window frames
CURRENT ROW delimiter, 269–270, 274, 276–278
default, 286
delimiters for, 269
as elements of window functions, 264
fast-track optimization cases, 272
FOLLOWING keyword, 269, 273–274
INTERVAL not supported, 276–277
offset functions, for, 286
offsets with RANGE, 276
offsets with ROWS, 273–274
partitions, relationship to, 269
PRECEDING keyword, 269–274, 277, 280–281
RANGE UNBOUNDED PRECEDING phrase, 279–280
RANGE unit with, 276–280
ROWS delimiters, 269–276
ROWS UNBOUNDED PRECEDING phrase, 270–271, 273, 281
spooling optimization, 278–279
ties in ordering value, RANGE vs. ROWS, 278
UNBOUNDED keyword, 269–272, 277, 281–286
year-to-date (YTD) calculation, 280–281
window functions
advantages of, 263
aggregates. See aggregate window functions
AVG OVER, 274–275
CUME_DIST OVER, 288–289
DENSE_RANK OVER, 282–283, 285, 294–296
elegance for calculations, 259
fast-track optimization plan, 271–272
filters with, 263–264
FIRST_VALUE OVER, 285–286, 511
frames of. See window frames
frequency calculations, 476–479
gaps problems, 291–292
GROUP BY with, 267–269
INTERVAL not supported, 276–277
inverse distribution functions, 289–291
island problems, 291–299
LAG OVER, 287, 296–299
LAST_VALUE OVER, 285–287, 511
LEAD OVER, 287, 291–292
legacy vs. 2012 versions, 259
limited to SELECT and ORDER BY clauses, 263
logical query-processing phases, 35–37
MAX OVER, 275–276
medians, finding with PERCENTILE_CONT, 369
moving average value computations with, 274
NTILE OVER, 282–283, 285
offset functions, 285–287
offsets with ROWS, 273–274
optimization of, 266–268
OVER clauses, 259, 263
packing interval solutions, 466–471
parallelism inhibited by certain, 179
window sets

window functions, continued
PARTITION BY with, 263–266
partitions with rankings, 281–285
PERCENTILE_CONT, 289–291, 369
PERCENTILE_DISC, 289–291
PERCENT_RANK OVER, 288–289
percents of grand totals with, 268–269
POC indexes with, 271–273, 284–285
rank distribution, 288–289
RANK OVER, 282–283, 285
ranking calculations with, 281–285
ROW_NUMBER OVER. See ROW_NUMBER OVER
window function
ROWS delimiters, 269–276
running total calculations with, 271–273
spooling optimization, 278–279
structures for, 264
SUM OVER. See SUM OVER window function
underlying results, exposure of, 263
vs. group functions, 259
windows of rows, 259
year-to-date (YTD) calculation, 280–281
window sets, 35–37
Windows Time service, 421–422
WITH RESULT SETS clauses, 573–575
WITHIN GROUP clauses, 327
WMAs (weighted moving averages), 514. See also
moving average value computations
write-ahead logging, 643

X

XML features
EVENTDATA as XML function, 579–580
EventData property for SQLCLR triggers, 616
FOR XML string concatenation, 317–319
PATH mode, 317, 531
TYPE directive, 317
XPacket data structure, 163

Y

YEAR function, 425
year-to-date (YTD) calculation, 280–281

Z

Z distribution, 487–488
About the authors

ITZIK BEN-GAN is a mentor for and co-founder of SolidQ. A SQL Server Microsoft MVP (Most Valuable Professional) since 1999, Itzik has delivered numerous training events around the world focused on T-SQL querying, query tuning, and programming. Itzik has authored several T-SQL books as well as articles for SQL Server Pro, SolidQ Journal, and MSDN. Itzik’s speaking activities include TechEd, SQLPASS, SQL Server Connections, SolidQ events, and various user groups around the world. Itzik is the author of SolidQ’s Advanced T-SQL Querying, Programming and Tuning, and T-SQL Fundamentals courses, along with being a primary resource within the company for its T-SQL-related activities.

DEJAN SARKA, MCT and SQL Server MVP, is an independent consultant, trainer, and developer focusing on database and business intelligence applications. His specialties are advanced topics like data modeling, data mining, and data quality. On these tough topics, he works and researches together with SolidQ and the Data Quality Institute. He is the founder of the Slovenian SQL Server and .NET Users Group. Dejan Sarka is the main author or coauthor of 11 books about databases and SQL Server, with more to come. He also has developed and is continuing to develop many courses and seminars for SolidQ and Microsoft. He has been a regular speaker at many conferences worldwide for more than 15 years, including Microsoft TechEd, PASS Summit, and others.

ADAM MACHANIC is a Boston-based SQL Server developer, writer, and speaker. He focuses on large-scale data warehouse performance and development, and he is the author of the award-winning SQL Server monitoring stored procedure sp_WhoIsActive. Adam has written for numerous websites and magazines, including SQLblog, Simple Talk, Search SQL Server, SQL Server Professional, CoDe, and VSJ. He has also contributed to several books on SQL Server, including SQL Server 2008 Internals (Microsoft Press, 2009) and Expert SQL Server 2005 Development (Apress, 2007). Adam regularly speaks at conferences and training events on a variety of SQL Server topics. He is a Microsoft Most Valuable Professional (MVP) for SQL Server, a Microsoft Certified IT Professional (MCITP), and an alumnus of the INETA North American Speakers Bureau.

KEVIN FARLEE has over 25 years in the industry, in both database and storage-management software. In his current role as a Storage Engine Program Manager on the Microsoft SQL Server team, he brings these threads together. His current projects include the SQL Server Project “Hekaton” In-Memory OLTP feature.
From technical overviews to drilldowns on special topics, get free ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in PDF, EPUB, and/or Mobi for Kindle formats.

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.
Now that you’ve read the book...

Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!