

# SQL Queries Joes 2 Pros

---

SQL Query Techniques  
For Microsoft SQL Server 2008

By  
Rick A. Morelan  
MCDBA, MCTS, MCITP, MCAD, MOE, MCSE, MCSE+I

©Rick A. Morelan  
All Rights Reserved 2009

ISBN: 1-4392-5318-8  
EAN: 978-1-4392-5318-2

Rick A. Morelan  
RMorelan@live.com

# Table of Contents

About the Author .....	9
Acknowledgements.....	9
Preface.....	10
Introduction.....	11
Skills Needed for this Book .....	12
About this Book .....	12
How to Use the Downloadable Companion Files.....	14
What this Book is Not.....	15
<b>Chapter 1. Data, Information, and Tables.....</b>	<b>16</b>
Single Table Queries.....	17
Basic Query Syntax.....	18
Exact Criteria .....	19
Pattern Criteria .....	21
Range Criteria .....	23
Querying Special Characters.....	26
Field Selection Lists.....	29
Anatomy of a SQL Query .....	30
Comparison Operators Used in Criteria.....	31
Lab 1.1: Basic Queries.....	32
Single Table Queries - Points to Ponder .....	33
Joining Tables .....	35
Inner Joins .....	35
Outer Joins .....	37
Left Outer Joins.....	38
Right Outer Joins .....	39
Full Outer Joins.....	40
Table Aliasing.....	41
Setting Aliases .....	42
Using Aliases .....	42
Lab 1.2: Joining Tables.....	45
Outer Joins - Points to Ponder .....	46
Bulk Copy Program (BCP) .....	47
Importing with BCP .....	47
Lab 1.3: Using BCP .....	50
Using BCP - Points to Ponder.....	51
Creating Tables .....	52
Inserting Data.....	53

Lab 1.4: Creating and Populating Tables.....	55
Creating and Populating Tables-Points to Ponder .....	57
Chapter Glossary.....	58
Chapter One - Review Quiz.....	59
Answer Key .....	61
Bug Catcher Game.....	63
<b>Chapter 2. Query Options .....</b>	<b>64</b>
Sorting Data .....	65
Sorting Single Table Queries .....	65
Sorting Multiple Table Queries .....	67
Sorting Data With Nulls.....	67
Sorting Hidden Results .....	68
Lab 2.1: Sorting Data.....	71
Sorting Data - Points to Ponder .....	74
Exploring Related Tables.....	75
Joining Two Tables.....	78
Joining Three Tables.....	79
Lab 2.2: Three Table Query .....	81
Three Table Query - Points to Ponder .....	82
Many-to-Many Relationships .....	83
Invoicing Systems.....	85
Lab 2.3: Many-To-Many Relationships.....	87
Many-to-Many - Points to Ponder .....	88
Chapter Two - Review Quiz .....	89
Answer Key .....	91
Bug Catcher Game.....	92
<b>Chapter 3. Null, Expression, and Identity Fields .....</b>	<b>93</b>
Working With Nulls.....	94
Nullable Fields in Database Tables.....	94
Inserting Nulls.....	96
Querying Nulls.....	96
Updating Nulls .....	97
Lab 3.1: Working With Nulls .....	99
Working With Nulls - Points to Ponder .....	101
Expression Fields .....	102
Calculated Fields.....	102
Field Functions.....	104
Aliasing Expression Fields .....	107
Sorting Expression Fields .....	108
Lab 3.2: Expression Fields.....	109
Expression Fields - Points to Ponder .....	111

Identity Fields .....	112
Analyzing Identity Fields.....	112
Creating Identity Fields.....	113
The Use of Identity Fields in Database Tables .....	115
Overriding Identity Fields.....	115
Lab 3.3: Identity Fields .....	121
Identity Inserts - Points to Ponder.....	122
Chapter Three - Review Quiz .....	123
Answer Key .....	125
Bug Catcher Game.....	126
<b>Chapter 4. Aggregating Data .....</b>	<b>127</b>
Using Group By .....	128
Group By With Sum .....	128
Group By With Count.....	129
Group By With Max .....	130
Counting Records vs. Counting Values .....	132
Multiple Level Grouping .....	135
Lab 4.1: Using Group By .....	139
Group By - Points to Ponder.....	141
Aggregation Criteria .....	142
The Having Clause.....	142
Where Clause in Aggregated Queries.....	147
Using Distinct .....	148
Lab 4.2: Filtering Aggregated Results .....	153
Filtering Aggregated Results - Points to Ponder.....	154
Aggregation in Stored Procedures .....	155
Creating the Aggregated query .....	155
Creating the Stored Procedure .....	156
Lab 4.3: Aggregation in Stored Procedures.....	159
Aggregation in Stored Procedures-Points to Ponder .....	160
Answer Key .....	164
Bug Catcher Game.....	166
<b>Chapter 5. Aggregation Strategies.....</b>	<b>167</b>
Duplicate Data .....	168
Sorting to Find Duplicates .....	168
Grouping to Find Duplicates.....	170
Lab 5.1: Finding Duplicates.....	174
Finding Duplicates - Points to Ponder .....	175
Combining Individual Rows With Aggregates.....	176
Aggregation Functions.....	176
Using the Over Clause .....	176

Lab 5.2: The Over Clause .....	187
The Over Clause - Points to Ponder.....	189
Chapter Five - Review Quiz .....	190
Answer Key .....	192
Bug Catcher Game.....	193
<b>Chapter 6. Top Records.....</b>	<b>194</b>
Top <i>n</i> Queries.....	195
Working With Tied Values.....	198
Top and Where.....	199
Deleting Top Data.....	200
Lab 6.1: Top N Queries .....	204
Top <i>n</i> Queries - Points to Ponder.....	206
Top Query Tricks.....	207
Top Queries With Variables .....	207
Customizing Top Queries .....	210
Lab 6.2: Top <i>n</i> Tricks.....	219
Top <i>n</i> Tricks - Points to Ponder .....	220
Chapter Six - Review Quiz .....	221
Answer Key .....	222
Bug Catcher Game.....	223
<b>Chapter 7. Ranking Functions .....</b>	<b>224</b>
The Rank Function.....	225
The Dense_Rank Function.....	228
Uses of Rank and Dense_Rank.....	231
Lab 7.1: Rank and Dense_Rank.....	237
Rank and Dense_Rank - Points to Ponder .....	239
Row_Number Function.....	240
Lab 7.2: Row_Number.....	244
Row_Number - Points to Ponder .....	245
Tiling Ranked Data .....	246
Lab 7.3: NTILE.....	250
NTILE - Points to Ponder .....	251
Chapter Seven - Review Quiz.....	252
Answer Key .....	254
Bug Catcher Game.....	255
<b>Chapter 8. Multiple Query Operators .....</b>	<b>256</b>
Multiple Queries With Similar Metadata.....	257
Union.....	257
Union All .....	263
Union Clause Rules.....	263
Lab 8.1: Union and Union All .....	267

Union and Union All - Points to Ponder .....	269
Using Intersect .....	270
Using Except.....	273
Lab 8.2: Intersect and Except.....	275
Intersect and Except - Points to Ponder .....	276
Chapter Eight - Review Quiz.....	277
Answer Key .....	278
Bug Catcher Game.....	279
<b>Chapter 9. Common Table Expressions (CTE).....</b>	<b>280</b>
Predicating on Expression Fields.....	281
Derived Tables .....	283
Common Table Expression (CTE).....	284
CTE and Expression Fields.....	288
CTE Field Aliasing .....	289
Lab 9.1: Common Table Expressions.....	290
Common Table Expressions - Points to Ponder .....	292
Advanced Clause Combinations .....	293
Using Compute .....	293
Using Compute By.....	297
Lab 9.2: Advanced Clause Combinations.....	299
Advanced Clause Combinations-Points to Ponder .....	301
Chapter Nine - Review Quiz.....	302
Answer Key .....	303
Bug Catcher Game.....	304
<b>Chapter 10. Data Recursion .....</b>	<b>305</b>
Single Table Hierarchies.....	306
Self Join Queries .....	309
Lab 10.1: Self Join Hierarchies.....	311
Self Join Hierarchies - Points to Ponder .....	312
Range Hierarchies .....	313
Lab 10.2: Range Hierarchies.....	321
Range Hierarchies - Points to Ponder .....	322
Recursive Queries .....	323
Lab 10.3: Recursive Queries.....	330
Recursive Queries - Points to Ponder .....	331
Chapter Ten - Review Quiz .....	332
Answer Key .....	334
Bug Catcher Game.....	335
<b>Chapter 11. Using Subqueries.....</b>	<b>336</b>
Basic Subqueries.....	337
Lab 11.1: Basic Subqueries.....	344

Basic Subqueries - Points to Ponder .....	346
Correlated Subqueries .....	347
Lab 11.2: Correlated Subqueries.....	357
Correlated Subqueries - Points to Ponder .....	358
Subquery Extensions.....	359
ALL.....	361
ANY and SOME .....	363
Lab 11.3: Subquery Extensions .....	368
Subquery Extensions - Points to Ponder.....	369
Chapter Eleven - Review Quiz .....	370
Answer Key .....	372
Bug Catcher Game .....	373
<b>Chapter 12. Table Data Actions.....</b>	<b>374</b>
Deleting vs. Truncating.....	375
Lab 12.1: Truncating and Deleting Data.....	381
Truncating and Deleting Data - Points to Ponder .....	382
Cross Apply .....	383
Lab 12.2: Cross Apply .....	391
Cross Apply - Points to Ponder.....	393
Outer Apply .....	394
Lab 12.3: Outer Apply .....	399
Outer Apply - Points to Ponder.....	402
Chapter Twelve - Review Quiz.....	403
Answer Key .....	405
Bug Catcher Game .....	406
<b>Chapter 13. The Merge Statement .....</b>	<b>407</b>
Introducing Merge .....	408
Upserting Multiple Records at Once.....	415
Lab 13.1: Using Merge .....	418
Using Merge - Points to Ponder.....	420
Merging Deletes.....	421
Merging Only Changes .....	425
Lab 13.2: MERGE Updating Options.....	430
MERGE Updating Options - Points to Ponder .....	431
Chapter Thirteen - Review Quiz .....	432
Answer Key .....	437
Bug Catcher Game .....	438
<b>Chapter 14. The Output Clause.....</b>	<b>439</b>
Output .....	440
Delete Actions With Output.....	440
Insert Actions With Output.....	441

Update Actions With Output .....	442
Using Output to Log Changes to a Table.....	445
Lab 14.1: Using Output.....	451
Using OUTPUT - Points to Ponder .....	453
Output Code Combinations.....	454
Output With Derived Tables .....	454
Output With MERGE Statements.....	457
Lab 14.2: OUTPUT Code Combinations.....	463
OUTPUT Code Combinations - Points to Ponder .....	466
Chapter Fourteen - Review Quiz .....	467
Answer Key .....	472
Bug Catcher Game .....	474
<b>Chapter 15. Common System Functions.....</b>	<b>475</b>
String Functions .....	476
Returning Parts of Strings .....	478
Changing Strings.....	480
Combining String Functions .....	483
Lab 15.1: String Functions.....	486
String Functions – Points to Ponder.....	488
Time Functions .....	489
Returning DateTime from Time Functions.....	489
Returning Numeric Results from Time Functions.....	491
Lab 15.2: Time Functions .....	498
Time Functions – Points to Ponder .....	499
Viewing Table and Field Metadata.....	500
Lab 15.3: Viewing Table and Field Metadata .....	506
Table and Field Metadata - Points to Ponder.....	507
Chapter Fifteen - Review Quiz .....	508
Answer Key .....	510
Bug Catcher Game .....	511
<b>Chapter 16. Next Steps for Aspiring SQL Pros .....</b>	<b>512</b>
<b>Index .....</b>	<b>516</b>

## About the Author

In 1994, you could find Rick Morelan braving the frigid waters of the Bering Sea as an Alaska commercial fisherman. His computer skills were non-existent at the time, so you might figure such beginnings seemed unlikely to lead him down the path to SQL Server expertise at Microsoft. However, every computer expert in the world today woke up at some point in their life knowing nothing about computers. They say luck is what happens when preparation meets opportunity. In the case of Rick Morelan, people were a big part of his good luck.

Making the change from fisherman seemed scary and took daily schooling at Catapult Software Training Institute. Rick got his lucky break in August 1995, working his first database job at Microsoft. Since that time, Rick has worked more than 10 years at Microsoft and has attained over 30 Microsoft technical certifications in applications, networking, databases and .NET development.

## Acknowledgements

As a book with a supporting web site, illustrations, media content and software scripts, it takes more than the usual author, illustrator and editor to put everything together into a great learning experience. Since my publisher has the more traditional contributor list available, I'd like to recognize the core team members:

**Editors:** Jessica Brown, Tom Ekberg, Irina Berger (MCTS)

**Illustrations:** Jessica Brown

**Technical Editor:** Jong Jin Lee (MCTS), Joel Heidal

**Technical Review:** Hugo Rosini, Doug Fritz, Gary Moore

**Software Design Testing:** Michael Baker (MCTS)

**Content Review:** Eugene Kim (MCTS), Scott Ekberg

**User Acceptance Testing:** Madhu Yadav

This experience has taught me that extraordinary talents are in people everywhere around us. For every need we've encountered, talented and motivated people have come out of the woodwork and showed up at just the right moment to support this effort.

Jessica Brown started out as my student in 2008 and her passion to help put accessible, user-friendly SQL education in the hands of students. This is a great gift to the *Joes 2 Pros* book series. She sees the good in everyone and every situation, and what I can say in one paragraph would only be a fraction of her value to these books.

Tom Ekberg has done so much for this book to support Jessica and myself and in fact the entire SQL Class of 2009. If you want someone talented who can make suggestions from the highest levels to the lowest details, then you have your man in Tom. He can wear the hats of creator, researcher, and double checker. I am so glad to have found Tom early on in the creation of this book.

Before this book started, I would have described Irina Berger as a dedicated successful graduate of my SQL class from years ago. She is an MCTS in SQL 2005 and a software designer. Today I can also tell you she uses her great understanding of technology at a project level. She made many suggestions that really raised the level of this book. With the use of a Windows Home Server, even when traveling out of the country, she continued to send her edits and kept things on track.

Hugo Rosini and Doug Fritz were just perfect for doing the technical editing work for this book. Both are accomplished in one or more programming languages, and they really were excited to add SQL Server to their portfolios. They both have a sharp eye and a goal-oriented nature. There is no better technical reviewer than someone who knows technology and wants to get every point made to complete their knowledge set. They truly left no stone unturned.

## Preface

How do you build a raging fire of learning from a single spark of curiosity? If you take that little spark and add some paper, a few twigs and a stick, eventually even the largest log can be tossed on to create a raging fire. You, too, can evolve from flame to firestorm, from Joe to Pro, when you take the steps outlined in this book. Regardless of your skill level, this is the ideal way to learn.

Viewing a large and unruly database that someone else wrote is like throwing a huge log directly onto a match. What if that large database, which included complex relations, started off as a smaller database? If you have completed the *Beginning SQL Joes 2 Pros* book you created new tables, data, logins, and other database objects. This book will add a few more items of your own creation to create a rich ground for practicing many different types of queries. You will learn how to build that large database one chapter at a time over the course of studying this book. You build it and then use it. When you are a part of building something, you comprehend each new level of complexity that is added. Afterward, you're able to stand back and say, "I built that and understand how it works!" Your *Joes 2 Pros* journey will soon make databases fun and familiar.

# Introduction

Does the following story sound familiar to you? The first SQL book I bought left me confused and demoralized at chapter one. Enrolling in my first class totally overwhelmed me and left me nearly hopeless – and with only a partial tuition refund. Progress was expensive and slow. Countless times I was tempted to give up.

After years of trial and error, I finally got into my groove. While grinding away at my own work with SQL Server, those key “ah-ha” moments and insights eventually came. What took me over five years of intense study is now a high tide that lifts my students to the same level in just a few months.

If you already know basic SQL syntax, this takes you to the next level. If you have no SQL coding experience, then first get into the groove of the *Beginning SQL Joes 2 Pros* book (ISBN 1-4392-5317-X). When you are at that level of SQL knowledge, you are ready for this book.

Each lesson and chapter builds sequentially until you know advanced query techniques used on the job and in most interviews. The labs have been created and delivered by me over several years of teaching SQL. If a lab offered one of my classrooms an exciting “ah-ha” experience with students leaning forward on every word and demo, it’s a keeper. However, when a lab caused more of a trance or tilted squint, it was discarded or revised with a better approach. The labs in this book are the end result, and each one consistently elicits “ah-ha” moments in my classes.

This book follows what students told me worked for them and launched their careers. This curriculum has helped many people achieve their career goals. If you would like to gain the confidence that comes with really knowing how to get things done, this book is your ticket.

Downloadable files help make this book a true learning experience. Answer keys, quiz games, and setup scripts will prepare you and your instance of SQL Server for practices that will hone your skills. The files can be found at [www.Joes2Pros.com](http://www.Joes2Pros.com). After you have run the right code several times, you are ready to write code and help others do the same by spotting errors in code samples. Each chapter’s interactive Bug Catcher section highlights common mistakes people make and improves your code literacy.

This book is an essential tool. When used correctly, you can determine how far and fast you can go. It has been polished and tuned for your use and benefit. In fact, this is the book I really wished was in my possession years ago when I was

learning about SQL Server. What took me years of struggle to learn can now be yours in only months in the form of efficient, enjoyable, and rewarding study.

## Skills Needed for this Book

Doing SQL Server queries requires basic knowledge of the SQL. What is SQL? SQL stands for Structured Query Language. You can do many things with the language like set up security, design tables, and get data in and out of tables. This book assumes you have been exposed to the concepts in the *Beginning SQL Joes 2 Pros* book, by past experience, or by completing the lessons. If you have but it's been awhile, then the first chapter is a review of lessons from the first book. Beyond that you need only be able to turn on your computer, click a mouse, type a little, and navigate to files and folders.

You should be able to install SQL Server on your computer. Your options are to either search the internet for a free download or buy a licensed copy. The official download site gets updated to a new location constantly. To get the most current installation steps go to [www.Joes2Pros.com](http://www.Joes2Pros.com).

For the free download you can search for “SQL Server Express 2008 download” and follow the instructions. It's a junior or light version. However, the preferred option is to search for “SQL Server 2008 Developer” and follow a link to one of the Microsoft download sites. Microsoft offers a real bargain for SQL students. For only \$50 you can install and use the fully-enabled Developer edition as long as you agree to use it only for your own learning and to create your own code. This is an outstanding deal considering that businesses generally spend \$10,000 to obtain and implement SQL Server. More on these options and installation instructions can be found in the `InstallingSQL2008Developer.wmv` and `InstallingSQL2008Express.wmv` download files.

## About this Book

By far the most common usage of SQL is to handle data. When you change or look at data, you are running queries. If you want to be a go-to person on queries and look forward to all such challenges, then this book strives for the same goal.

*Beginning SQL Joes 2 Pros* began in the summer of 2006. The project started as a few easy-to-view labs to transform the old, dry text reading into easier and fun lessons for the classroom. The labs grew into stories. The stories grew into chapters. In 2008, many people whose lives and careers had been improved

through my classes convinced me to write a book to reach out to more people. In 2009 the first book began in full gear until its completion (*Beginning SQL Joes 2 Pros*, ISBN 1-4392-5317-X).

It's easy to boost the page count without adding content. Wide margins that go in an one or two inches or more from every edge and giant indexes can bloat the size of a 300 page book to 500 pages or more without new content. Many of the publishers that said no to me cited big books give them a wide spine on the book shelf. The only way to show them quality wins out is if enough people ask for this book.

Maybe the publishers are right about hooking the consumer with page count and padding. People buy on page count and read on content, but how often are bad, bloated books returned? Perhaps readers are now ready to get the most from their investment. Trim off the fat and get what you deserve for less. When the world is ready, it will reflect in this book's success.

There are other ways to deliver good teaching without overdosing you on ink. How many words would it take to describe your favorite shirt? Would you rather write them all out or show someone with pictures and videos? Every point made in *Joes 2 Pros* is demonstrated with life-like examples using real databases, thus the downloadable files are essential tools in your SQL Server learning. The instructional videos will further enhance your learning experience.

There was a time when the term "boot camp" meant something admirable in the training realm. Too many programs today prey on students' dreams while delivering little in return. These incentives rule stronger than ever in the IT training business model. A worst-case example was a school where I briefly taught. It charged students the highest tuition rate I've ever encountered while paying instructors the least amount, yet spent far more on advertising in the form of radio spots to lure unsuspecting students seeking to gain IT skills. That school also has the lowest rate of students successfully completing the program than any I have encountered. *Beginning SQL Joes 2 Pros* was written to change this trend.

It is time for books and schools to compete on quality and effectiveness and not a mass marketing push. The better this book does, the more we will get their attention and what people really need and are ready for.

To put it simply, there is a recipe for success. You can choose your own ingredients. Just learn the lesson, do the lab, view Points to Ponder, and play the review game at the end of each chapter.

Most of the exercises in this book are designed around proper database practices in the workplace. The workplace also offers common challenges and process changes over time. For example, it is good practice to use numeric data for IDs. If

you have ever seen a Canadian postal code (zip code), you see the need for character data in relational information. You will occasionally see an off-the-beaten-path strategy demonstrated so you know how to approach a topic in a job interview or workplace assignment.

Some students like the Read-View-Do approach so they know the concepts of what they are about to see. Others like to View-Read-Do approach so they can visualize the concepts as they read.

In the Read-View-Do approach you read the chapter, view the videos, and do the challenges in the videos. For example, let's say you are on chapter 2. You will read the pages for Chapter 2, view the Lab 2.1 video, and do the video Lab 2.1 challenge(s) at the end of the video. In the View-Read-Do approach for Chapter 2 you will watch the Lab 2.1 video, skip the Skill Check until later, read the book, and do the video Lab 2.1 challenge(s).

## How to Use the Downloadable Companion Files

Clear content and high-resolution multimedia videos coupled with code samples will take you on this journey. To give you all this and save printing costs, all supporting files are available with a free download from [www.Joes2Pros.com](http://www.Joes2Pros.com). The breakdown of the offerings from these supporting files is listed below:

**Training videos:** To get you started, the first three chapters are in video format for free downloading. Videos show labs, demonstrate concepts, and review Points to Ponder along with tips from the appendix. Ranging from 3-15 minutes in length, they use special effects to highlight key points. You can go at your own pace and pause or replay within lessons as needed.

**Answer keys:** The downloadable files also include an answer key. You can verify your completed work against these keys. Another helpful use is these coding answers are available for peeking if you get really stuck.

**Resource files:** If you are asked to import a file into SQL Server, you will need that resource file. Located in the resources sub-folder from the download site are your practice lab resource files. These files hold the few non-SQL script files needed for some labs.

**Lab setup files:** SQL Server is a database engine and we need to practice on a database. The Joes 2 Pros Practice Company database is a fictitious travel booking company whose name is shortened to the database name of JProCo. The scripts to set up the JProCo database can be found here.

**Chapter review files:** Ready to take your new skills out for a test drive? We have the ever popular Bug Catcher game located here.

## What this Book is Not

This book will start you off on the cornerstones of the language behind SQL Server. It will not cover every keyword, as you will get many of them as you work and become a SQL Server expert.

This is not a memorization book. Rather, this is a skills book to make preparing for the certification test a familiarization process. This book prepares you to apply what you've learned to answer SQL questions in the job setting. The highest hopes are that your progress and level of SQL knowledge will soon have business managers seeking your expertise to provide the reporting and information vital to their decision making. It's a good feeling to achieve and to help at the same time. Many students commented that the training method used in *Joes 2 Pros* was what finally helped them achieve their goal of certification.

When you go through the *Joes 2 Pros* series and really know this material, you deserve a fair shot at SQL certification. Use only authentic testing engines drawing on your skill. Show you know it for real. At the time of this writing, MeasureUp<sup>®</sup> at <http://www.measureup.com> provides a good test preparation simulator. The company's test pass guarantee makes it a very appealing option.

# Chapter 1. Data, Information, and Tables

Most of my students laugh when they hear me say the prerequisite for learning in my classes is the ability to click a mouse and breathe oxygen. A beginning class should be welcoming for beginners. That is the reason my first book called *Beginning SQL Joes 2 Pros* was written. In this book, Chapter 1 will be a beginner quick refresher on data, information, and tables. This is just in case you decided to start with this book and are a beginner, or you completed the first book but want a refresher. Many pieces of the first book are interspersed in this chapter. If you are an intermediate, treat Chapter 1 as a warm up and get ready for some fun.

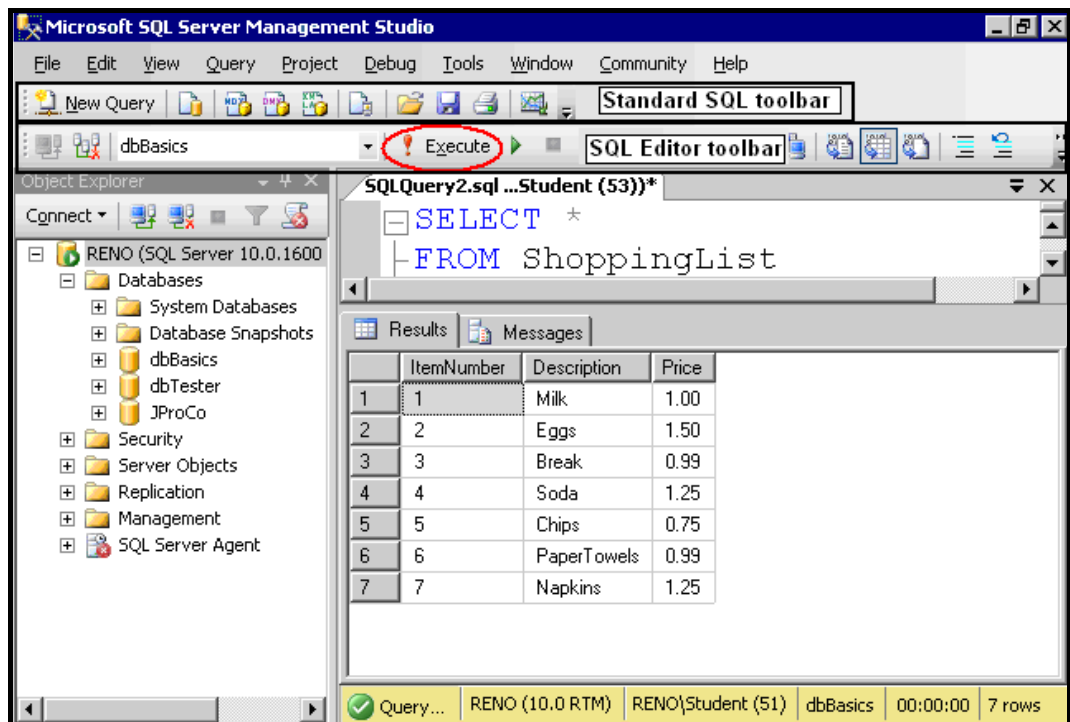
Each chapter will include an instruction for the setup script you need to run in order to follow along with the chapter examples. The setup scripts give you the freedom to practice any code you like – including changing or deleting data, or even dropping entire tables or databases. Afterwards you can rerun the setup script (or run the setup script when you reach the next section) and all needed objects and data will be restored. This process is also good practice for beginners – these are typical tasks done frequently when working with SQL Server. If you need a refresher on installing SQL Server, see the resources mentioned on page 12, including the two installation videos (InstallingSQL2008Developer.wmv and InstallingSQL2008Express.wmv). In the first book (*Beginning SQL*), we created a C:\Joes2Pros folder to hold all your resource files loaded from the website. If you don't already have a C:\Joes2Pros folder, please create one now since you will need it for this chapter's exercises.

**READER NOTE:** Please run the script *SQLQueriesChapter1.0Setup.sql* in order to follow along with the examples in the first section of Chapter 1. All scripts mentioned in this chapter may be found at [www.Joes2Pros.com](http://www.Joes2Pros.com).

## Single Table Queries

When you want information from your database, you write a query. A query is a request for data from a database. In English you might say, “Show me the information!” In SQL you say, “SELECT.” It’s easy to spot a query statement since it always starts with the word SELECT.

The word SELECT means you want to see all the information displayed. The asterisk (\*) is handy if you don’t know the names of the fields but want all of them to be displayed. The keyword FROM chooses the table you’re querying for the information. Run your query by hitting the F5 key on your keyboard or clicking the execute button in the SQL Editor toolbar (circled below in red).



**Figure 1.1** A query for all records and all fields from the ShoppingList table.

## Basic Query Syntax

Now is a good time to point out you can put words in your query window that have nothing to do with SQL. You could write non-SQL words and notes like “I wrote this query today” above your SELECT statement. If you do, you must tell SQL to ignore this text, since it’s not intended to be run as code. In fact, it should be non-executing code known as comments. To make comments, you need to begin the line with two hyphens. Here is an example:

```
--This is my latest query  
SELECT * FROM Location
```

The code above runs fine in the JProCo database. The first line is ignored by SQL because of the double hyphens. It’s there only for your benefit. This is a useful way to make notes that later provide you or your team with key hints or details explaining what the code is attempting to accomplish.

Table names can optionally have square brackets around them. This does not change your result set. The two queries below operate identically. One uses square brackets while the other does not:

```
SELECT * FROM ShoppingList  
SELECT * FROM [ShoppingList]
```

When coding, you rarely use square brackets because it means extra typing. The only time bracketing is critical is when you can’t tell where a table name starts or stops. On the other hand, the dbBasics database has two identical tables named “ShoppingList” and “Shopping List.” The latter contains a space in its name, which is generally a bad naming practice for any database object.

In the case of “Shopping List”, you must use a delimiter, such as square brackets. Without these delimiters, SQL Server will think the table is named “Shopping” and you have a command named “List” which it does not recognize.

You can put square brackets around any table. In fact, automatically generated code always creates these delimiters. The only time you need to do this is when table names are not obvious to SQL Server (e.g., when a table name is the same as a SQL keyword).

Delimiting table names helps us in another way. It would be a bad idea to name your tables after known keywords. For example, a table named “From” would look like this:

```
SELECT * FROM From
```

The vocabulary of SQL Server has grown over the years, and new keywords are continually added. Take an example where a company database that keeps track of charity grants is named “Grant.” The company upgrades to a newer version of SQL Server where Grant is a keyword used to administer permissions. The following code would create problems:

```
SELECT * FROM Grant
```

Without delimiting the Grant table name, SQL Server tries to use “Grant” as a keyword. You can resolve table name and keyword conflicts by using delimiters. By putting square brackets around the table name of Grant, you tell SQL Server it’s a table name and not the GRANT keyword we will study in Chapter 10.

```
SELECT * FROM [Grant]
```

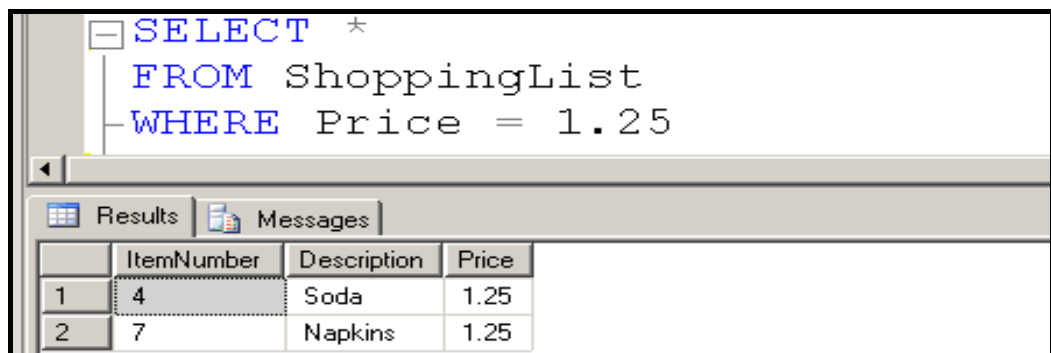
Square brackets will work on all table identifiers. You can put square brackets around the [Employee] table if you like. In reality, you only need them occasionally.

This also works for fields. For example, the [Grant] table has a field named “State.” In the 2008 version of SQL Server, STATE became a keyword. Thus, we must write it as [State] when including that field in our code. We will discuss field selection lists later on in this chapter.

## Exact Criteria

As we have seen so far, the two keywords most commonly used in queries are SELECT and FROM – these will be required in nearly all the queries you will use to retrieve data during your SQL Server career. The most common optional keyword is WHERE.

We want to limit our result set based on the Price field being exactly \$1.25. We add this specific criteria using the WHERE clause (Figure 1.2).



**Figure 1.2** The WHERE clause limits the number of records in your result set.

The WHERE clause is perfect for filtering your information. The amount of data in the ShoppingList table is still seven records, but the number of records in our result set is now just two.

The WHERE clause can filter any type of data. If we wanted to see all employees named David in the JProCo database we can use the WHERE clause. When using character data, we must use single quotation marks as seen in this query:

```
SELECT *  
FROM Employee  
WHERE FirstName = 'David'
```

The above query uses a WHERE clause to filter on the FirstName field to look for all employees with the first name of David. We still have 12 records of data stored in the Employee table. The WHERE clause shows us just the records in our result set that we asked to view.

WHERE expects a logical statement to evaluate each record. The logical statement `FirstName = David` is called a *predicate*. By predicating on FirstName, you filter your result set.

When you put an exclamation point before an equal sign it means “not equals.” Exact matches are filtered out of the result set. If you have used previous versions of SQL Server, you may be familiar with the `<>` operator as meaning “Not Equals.” That is still true today with the code you see below:

```
SELECT *  
FROM Employee  
WHERE FirstName < > 'David'
```

The risk in using `<>` is that it looks too much like an HTML or XML tag. Thus, when SQL needs to talk to other languages, it’s better to use `!=` to signify “not equal to.”

Using the equal sign gives you an exact match. What if you wanted to look for all the first names of Lisa or David? One option is to use the code below:

```
SELECT *  
FROM Employee  
WHERE FirstName = 'Lisa'  
OR FirstName = 'David'
```

You can specify only one exact match after the equal operator. To query for multiple exact matches, you would need multiple equal signs. Alternatively, you can list a set of exact matches in one criterion by using the *IN* operator. The code below is another way to find the same result set:

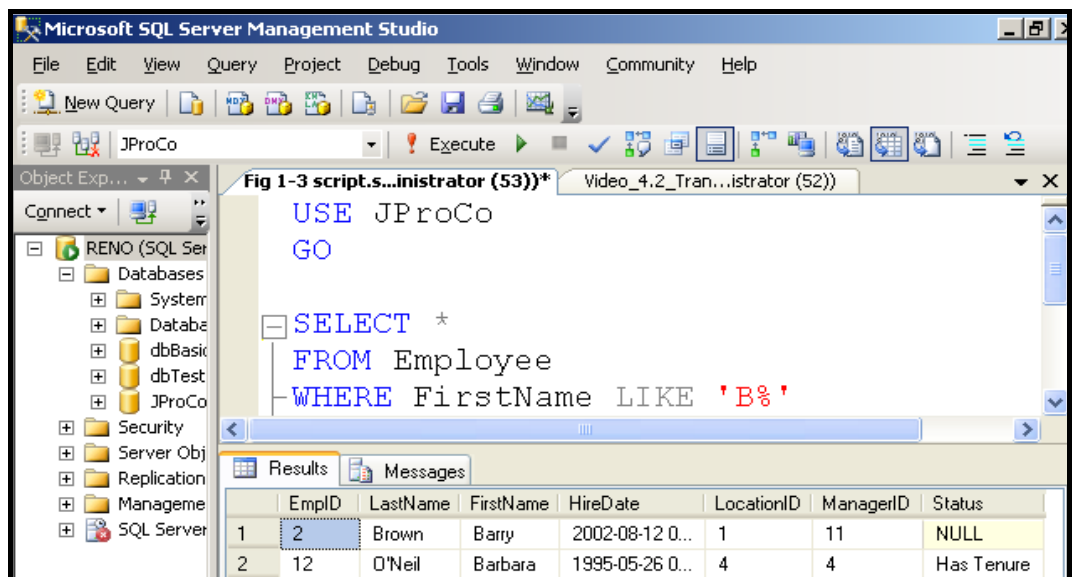
```
SELECT *
FROM Employee
WHERE FirstName IN ('Lisa', 'David')
```

## Pattern Criteria

What do the names Barry and Bo have in common? They both start with the letter B. Yes, there are other similarities, but let's go with the most obvious. If you use the = operator with the letter B, it wouldn't find either name in the Employee table. It would look for the name "B," which is only one letter long. Your query would return an empty result set. We need to combine an approximate operator with something called a *wildcard*.

The operator that allows you to use approximate predicates is *LIKE*. The *LIKE* operator allows you to do special relative searches to filter your result set.

To find everyone whose first name starts with the letter B, you need "B" to be the first letter. After the letter B you can have any number of characters. Using B% in single quotes after the *LIKE* operator gets all names starting with the letter B (as shown below in Figure 1.3).



**Figure 1.3** Using the approximate operator *LIKE* allows for a wildcard in your predicate.

This is a good time to point out that SQL does not care whether your name is “Barry” or “barry”. *SQL is not case sensitive unless you go out of your way to change its default setting.*

The % wildcard symbol represents any number of characters. Let’s find all first names that end in the letter A. By using the percentage “%” symbol with the letter ‘A’, you achieve this goal using the code sample below:

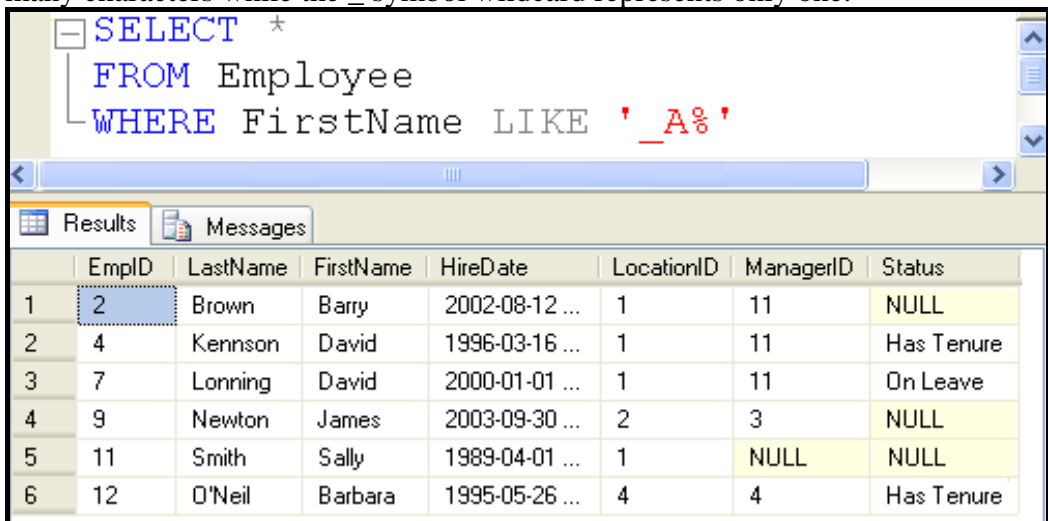
```
SELECT *  
FROM Employee  
WHERE FirstName LIKE '%A'
```

Lisa and Barbara both end in the letter A. In this example, a capital A was used and it found all first names ending in A – even if they were lower case, since SQL Server is case insensitive.

Lisa has three characters before the ending letter A, while Barbara has six. The % wildcard can mean three characters. It can also represent one, nine, one hundred, or even zero characters. If your first name was just “A” then you would also appear in this result set.

The next goal is to find FirstName records that have the letter A as the second letter. We want exactly one character of any type followed by an A, then any number of letters afterwards. A wildcard of exactly one character is represented by the underscore “\_” symbol.

By asking for one character before the letter A and any amount afterward, we find David, James, and others (Figure 1.4). The % symbol wildcard can represent many characters while the \_ symbol wildcard represents only one.



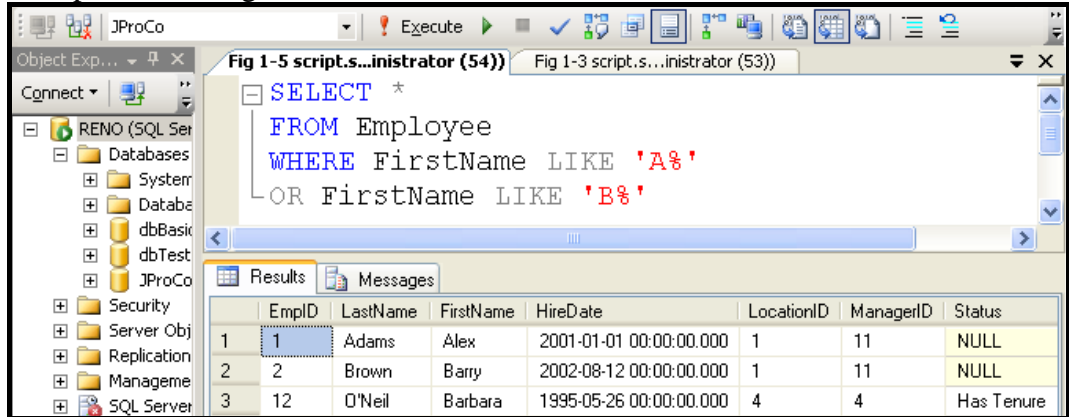
```
SELECT *  
FROM Employee  
WHERE FirstName LIKE '_A%'
```

	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	2	Brown	Barry	2002-08-12 ...	1	11	NULL
2	4	Kennson	David	1996-03-16 ...	1	11	Has Tenure
3	7	Lonning	David	2000-01-01 ...	1	11	On Leave
4	9	Newton	James	2003-09-30 ...	2	3	NULL
5	11	Smith	Sally	1989-04-01 ...	1	NULL	NULL
6	12	O'Neil	Barbara	1995-05-26 ...	4	4	Has Tenure

**Figure 1.4** Using the underscore wildcard to find exactly one character.

## Range Criteria

If you want to find all `FirstName` values starting with the letters A or B, you can use two predicates in your `WHERE` clause. You need to separate them with the *OR* operator. In Figure 1.5, we find records where `FirstName` starts with A or B.



**Figure 1.5** Using the *OR* operator to find `FirstName` values starting with the letters A or B.

Finding names beginning with A or B is easy. This works well until you want to search using a range of letters, such as A-K shown in the example below:

```
SELECT *
FROM Employee
WHERE FirstName Like 'A%'
OR FirstName Like 'B%'
OR FirstName Like 'C%'
OR FirstName Like 'D%'
OR FirstName Like 'E%'
OR FirstName Like 'F%'
OR FirstName Like 'G%'
OR FirstName Like 'H%'
OR FirstName Like 'I%'
OR FirstName Like 'J%'
OR FirstName Like 'K%'
```

This lengthy query (above) will find `FirstName` values beginning with A-K. But if you need a range of letters, the `LIKE` operator has a nicer option. Since we're looking for the first letter to be within a range, we specify the range in square

brackets. The wildcard after the brackets allows the FirstName to contain any number of characters after the first letter.

This range would not work if your LIKE was changed to an exact operator. The following code will not return any records to your result set:

```
--Bad query (it won't error but returns no records)
SELECT *
FROM Employee
WHERE LastName = '[A-K]%'
```

A range of characters can be found using LIKE and your desired characters (e.g., A-K) inside square brackets. The wildcard is considered a string pattern and must be enclosed in single quotes. (Strings and string functions will be studied in Chapter 15). Simply put, it's the starting letter followed by a hyphen and then the ending letter of your range.

```
SELECT *
FROM Employee
WHERE LastName LIKE '[A-K]%'
```

Notice you get Adams in your result set. This is because A falls within the [A-K] range. The same is true for Kendall. There is a similar trick you can play with number ranges.

If you look at the Grant table you will notice we get amounts as low as \$4,750 and as high as \$41,000 for the Amount field (Figure 1.6).

The screenshot shows the Microsoft SQL Server Management Studio interface. The query window displays the following SQL statement:

```
SELECT * FROM [Grant]
```

The Results pane shows the following data:

	GrantID	GrantName	EmpID	Amount
1	001	92 Pur_Scents %% team	7	4750.00
2	002	K_Land fund trust	2	15750.00
3	003	Robert@BigStarBank.com	7	18100.00
4	004	Norman's Outreach	NULL	21000.00
5	005	BIG 6's Foundation%	4	21000.00
6	006	TALTA_Kishan International	3	18100.00
7	007	Ben@MoreTechnology.com	10	41000.00
8	008	www.@-Last-U-Can-Help.com	7	25000.00
9	009	Thank you @.com	11	21500.00
10	010	Call Mom @Com	5	7500.00

Figure 1.6 All records from the [Grant] table.

We have multiple grants that are over \$20,000 in the Amount field. In the following query, we use the “greater than” operator to find amounts over 20000:

```
SELECT *
FROM [Grant]
WHERE Amount > 20000
```

We have multiple grants that are under \$20,000 in the amount field. The following query uses the “less than” operator to find amounts under 20000:

```
SELECT *
FROM [Grant]
WHERE Amount < 20000
```

We need to be careful when looking for amounts over \$21,000 (Figure 1.7) because we have some grants that are exactly \$21,000. Using greater than > will not include this amount, but greater than or equal to >= will include the matching amount of \$21,000.

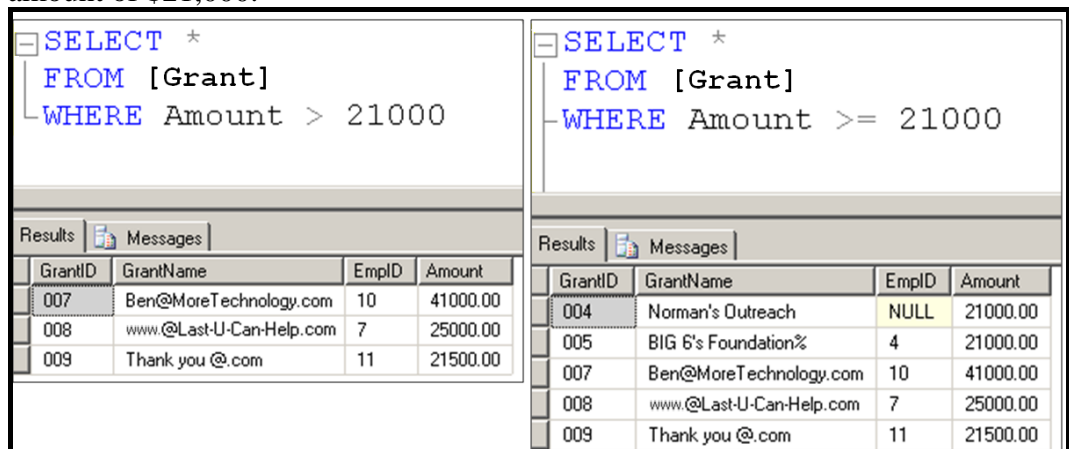
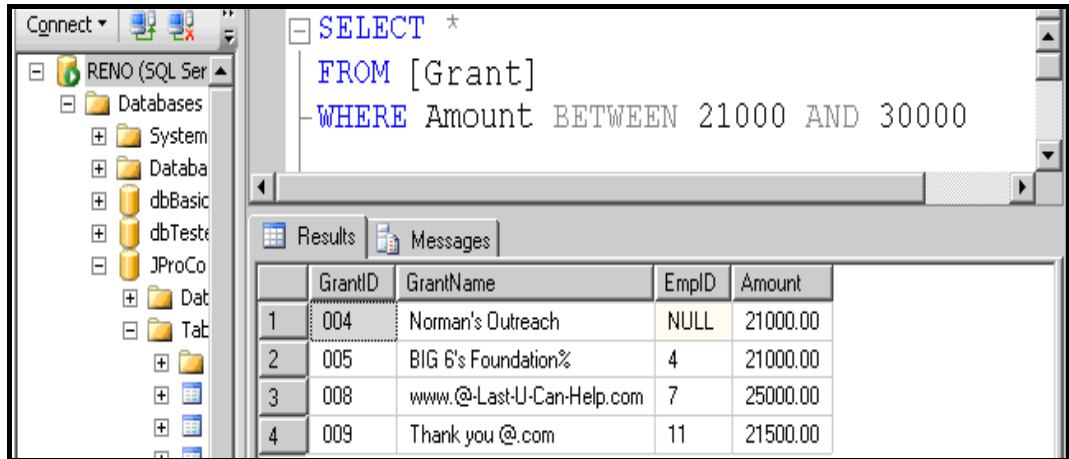


Figure 1.7 Result sets for “greater than” vs. “greater than or equal to” operators.

When someone asks you to pick a number between one and 10, what are valid answers? Zero would be out of range. How about an edge case answer of one? Is one between one and 10? Yes! The same is true with the BETWEEN operator in SQL Server.

After the WHERE, you can use the BETWEEN operator with the AND operator to specify two numbers that define your range. When we look for amounts between 21000 and 30000, we get four records in our result set (Figure 1.8).



**Figure 1.8** Using BETWEEN with AND to get a range of values.

Notice that two of our results are exactly 21,000. BETWEEN offers you results that are inclusive of the numbers specified in your predicate.

## Querying Special Characters

Earlier we learned about two special characters called wildcards. We can use the percentage sign % or the underscore \_ in relative searches. There is a grant “92 Purr\_Scents %% team” which has a percentage symbol in the name. We have other grants with percentages in their names. How do you search for a percent sign with two wildcards on either side? In the query below, it would appear to SQL Server that you’re looking for three wildcards:

```
Bad query pattern logic (finds all records)
SELECT *
FROM [Grant]
WHERE GrantName LIKE '%%%'
```

We have three special characters and no literal percent symbol. The square brackets will help here. Take the wildcard you want to use as a literal percentage symbol and enclose it inside square brackets. You see two grants having a percentage symbol within their names (Figure 1.9). In this example the square brackets give you the literal percentage symbol.

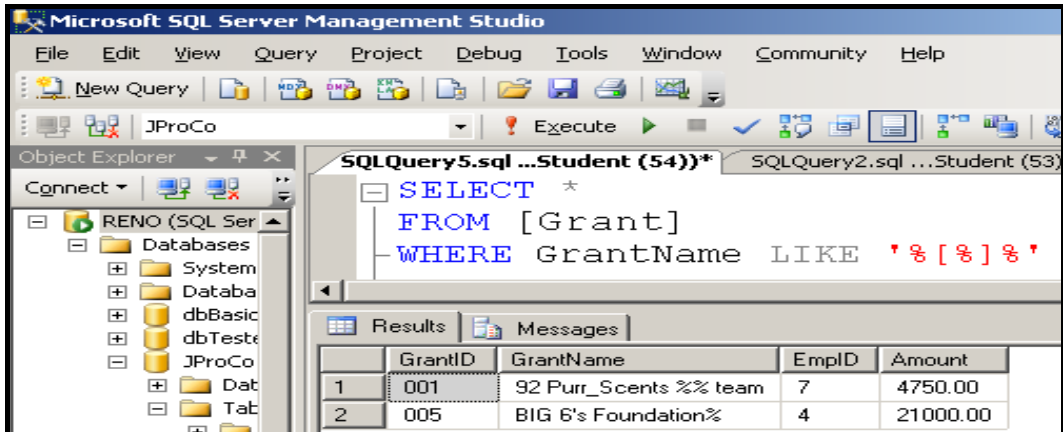


Figure 1.9 Finding an actual % sign using a relative search predicate.

We have a grant called “K\_Land fund trust” with an actual underscore in the name. We have other grants with underscores, as well. How do you search for an underscore sign with a wildcard on each side? The ineffective query below displays all records that are at least one character long:

**Bad query logic finds all records with one or more characters**

```
SELECT *
FROM [Grant]
WHERE GrantName LIKE '%_%'
```

This code shows three special characters and no literal underscore. Again, we need to put the character that we want to find inside square brackets.

Your code now shows the three grants having underscores in their names. The square brackets tell SQL Server we are looking for a literal underscore character.

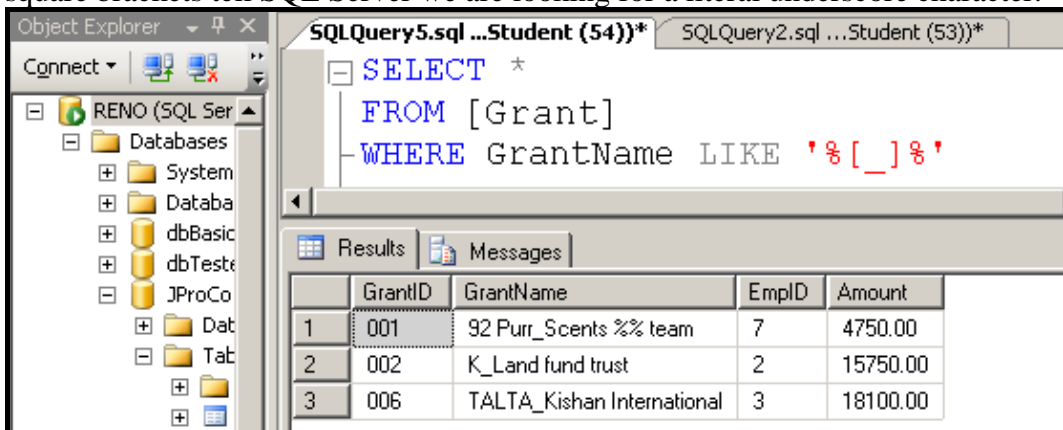


Figure 1.10 Using square brackets around the underscore finds all values with a literal underscore.

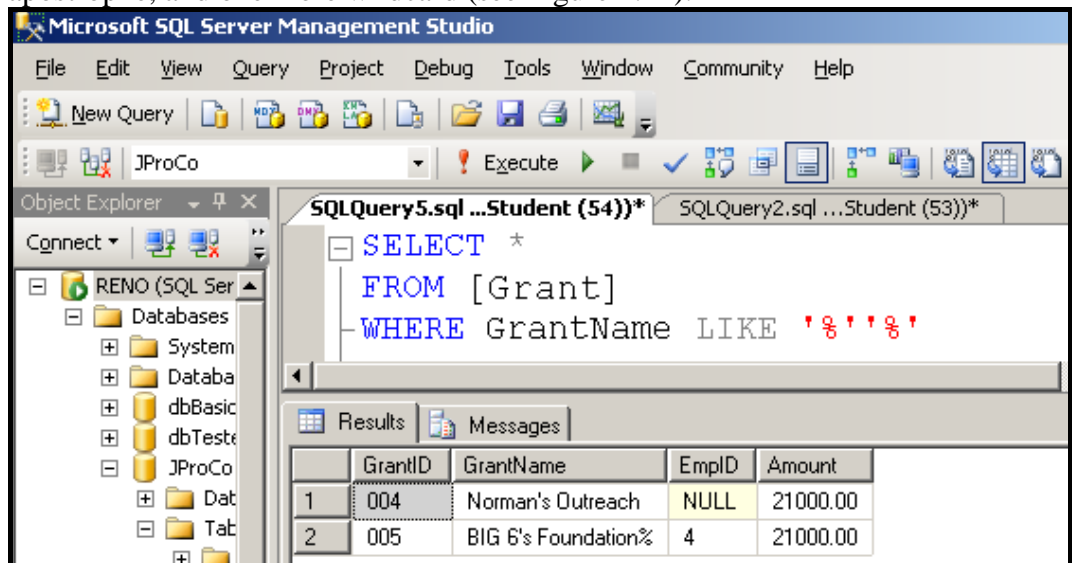
Everything inside single quotes after the LIKE evaluates every record to give you your final result set. The first single quote starts the string and it ends with the second single quote. Anything before the first single quote and after the second single quote is not part of the search string. The single quote marks beginning and end – or *delimits* – the pattern you are searching.

A new challenge arises here. What if you want to find grants with an apostrophe (single quote) in the name, such as Norman's Outreach? The following query produces a syntax error.

**Bad query results in an error**

```
SELECT *
FROM [Grant]
WHERE GrantName LIKE '% %'
```

The problem here is that SQL Server assumes the predicate is complete after the second single quote. SQL Server sees everything after the second single quote as an error in your SQL code. To negate the special meaning of the single quote, precede it with another single quote. Inside the outer single quotes, SQL Server now recognizes the sequence you want to search: one wildcard, one single apostrophe, and one more wildcard (see Figure 1.11).

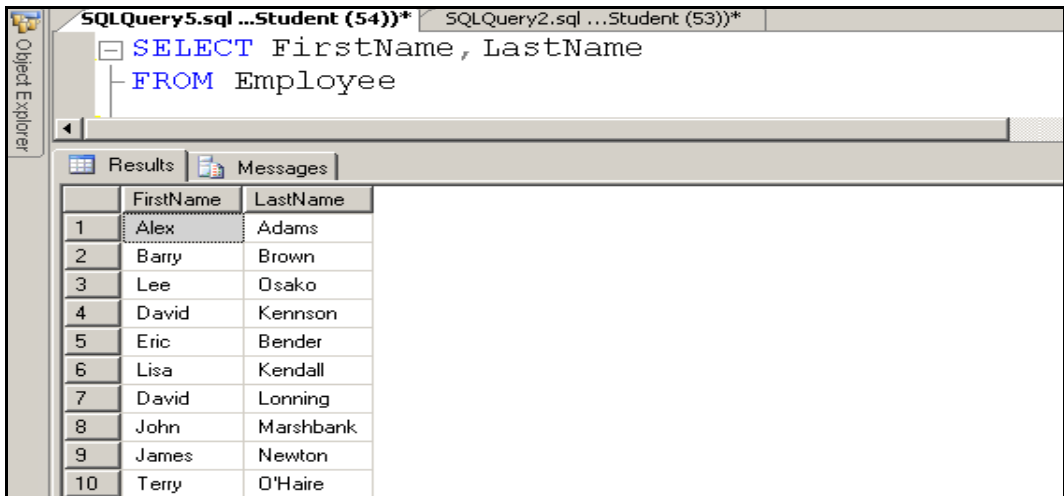


**Figure 1.11** Using two single quotes will filter for grant names containing an apostrophe.

You now have two records with a single quote in your result set. To view all names without a single quote you would simply change the LIKE to NOT LIKE in the WHERE clause (WHERE GrantName NOT LIKE '% ' %').

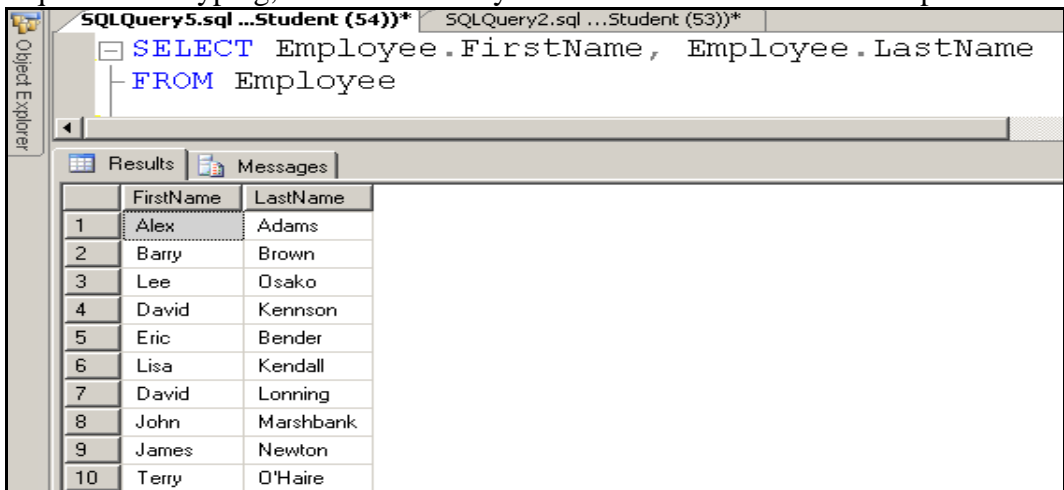
## Field Selection Lists

Thus far our queries have used the asterisk \* symbol right after the SELECT clause. This is both handy and common for looking at information. The asterisk frees you from specifying the names of the fields. One drawback to this method is you get all fields. Sometimes you want only a few fields in your result set. By listing the FirstName and LastName fields separated by a comma, we get just two fields in our result set (Figure 1.12).



**Figure 1.12** Changing your field select list to show just FirstName and LastName fields.

Optionally, you can use the two-part name of the field by listing the table identifier and then the field identifier separated by a period (Figure 1.13). This requires extra typing, which we show you how to avoid later in this chapter.



**Figure 1.13** Field select list using the two-part naming convention of *TableName.FieldName*.

Using two-part names for fields uses the `TableName.FieldName` syntax. `Employee.FirstName` and `Employee.LastName` (Figure 1.13) gives us the same results as just specifying `FirstName` and `LastName` (Figure 1.12).

When field names clash with an existing keyword, it's wise to put the square brackets around the table or field (e.g., the `[Grant]` table, the `[State]` field in the `Location` table) to tell SQL Server you're referring to the object and not the keyword command. We touched on this earlier in our discussion of square bracket delimiters and special characters. In some cases, SQL Server may run your query without an error. For example, the following code will probably run in your instance of SQL Server and return data.

```
SELECT Street, City, State  
FROM Location
```

It simply turns the “State” keyword blue (i.e., meaning it recognizes `State` as a command instead of an object). But a later version of SQL may not be so forgiving. Also, if you are sharing code in a team environment, it may be confusing for team members to see a table or field name in blue font instead of black. A best practice is to always bracket object names you know are keywords in SQL Server.

## Anatomy of a SQL Query

Remember that query keyword order must be followed in your SQL code. Here is the full breakdown of the anatomy of a basic query.

<b>USE JProCo</b>	<b>Choose your DATABASE</b>
<b>GO</b>	<b>Finish last statement</b>
<b>SELECT Firstname, Lastname</b>	<b>Choose your Field (s)</b>
<b>FROM Employee</b>	<b>Choose your Table(s)</b>
<b>WHERE Firstname = 'Lisa'</b>	<b>Filter the Result Set</b>

## Comparison Operators Used in Criteria

The following table lists the Transact-SQL (called T-SQL for short) comparison operators. Comparison operators evaluate expressions in order to display the precise data you want included in your result set.

**Table 1.1** Comparison operators used in T-SQL queries.

<b>Operator</b>	<b>Description</b>
=	Equality for one value
<>	Non-equality (deprecated)
!=	Non-equality
<	Less than
<=	Less than or equal to
!<	Not less than
>	Greater than
>=	Greater than or equal to
!>	Not greater than
BETWEEN	Between two specified values
IN	Equality for enumeration of values

## Lab 1.1: Basic Queries

**Lab Prep:** Each lab has one or more Skill Checks. Start with Skill Check 1 and proceed until you reach the Points to Ponder section. Before you can begin the lab you must have SQL Server installed and run the SQLQueriesChapter1.1Setup.sql script. Since this is your first lab, please make sure you have viewed the video on how to set up a typical lab called SQLQueriesSetupLabSteps.wmv. This video shows you the steps involved in setting up all the labs in this book.

**Skill Check 1:** Write a query that displays a result set of all records from the CurrentProducts table (inside the JProCo database) with a RetailPrice of 200 or less. When you are done, your result should resemble Figure 1.14 below.

	ProductID	ProductName	RetailPrice	OriginationDate	ToBeDeleted	Category
1	1	Underwater Tour 1 Day West Coast	61.483	2006-08-11 13:33:09.957	0	No-Stay
2	2	Underwater Tour 2 Days West Coast	110.6694	2007-10-03 23:43:22.813	0	Overnight-Stay
3	3	Underwater Tour 3 Days West Coast	184.449	2009-05-09 16:07:49.900	0	Medium-Stay
4	7	Underwater Tour 1 Day East Coast	80.859	2007-04-07 08:25:43.233	0	No-Stay
5	8	Underwater Tour 2 Days East Coast	145.5462	2005-06-11 09:52:12.910	0	Overnight-Stay

Query executed successfully. RENO (10.0 RTM) RENO\Student (51) JProCo 00:00:00 197 rows

Figure 1.14 Skill Check 1 should produce 197 records.

**Skill Check 2:** Find all the records from the CurrentProducts table that have the word Canada in the ProductName. Show all fields from the CurrentProducts table. When you are done, your result should resemble Figure 1.15 below.

	ProductID	ProductName	RetailPrice	OriginationDate	ToBeDeleted	Category
1	19	Underwater Tour 1 Day Canada	85.585	2004-04-18 16:12:07.293	0	No-Stay
2	20	Underwater Tour 2 Days Canada	154.053	2006-10-30 10:10:51.630	0	Overnight-S
3	21	Underwater Tour 3 Days Canada	256.755	2005-10-14 21:07:11.843	0	Medium-Sta
4	22	Underwater Tour 5 Days Canada	342.24	2001-08-18 05:51:42.250	0	Medium-Sta

Query executed successfully. (local) (10.0 RTM) RENO\Student (51) JProCo 00:00:00 96 rows

Figure 1.15 Skill Check 2 produces 96 records.

**Skill Check 3:** Grant is a table in the JProCo database. Show all the Grant records with Amount values between 21000 and 30000. Show only the GrantName and Amount fields. When you are done, your result should resemble the figure below.

	GrantName	Amount
1	Norman's Outreach	21000.00
2	BIG 6's Foundation%	21000.00
3	www.@-Last-U-Can-Help.com	25000.00
4	Thank you @.com	21500.00

Figure 1.16 Skill Check 3 produces four records.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab1.1\_BasicQueries.sql.

## Single Table Queries - Points to Ponder

1. SQL Server uses the Transact Structured Query Language (T-SQL).
2. A query is written in the T-SQL language and is a request for information from a database.
3. Database context refers to which database you are running the current query against.
4. The FROM clause tells SQL which table or tables you are interested in viewing.
5. Always maintain the following query keyword order: SELECT, FROM and WHERE.
6. An asterisk \* placed directly after SELECT means you want to view all available fields.
7. If you want to see only a subset of fields in your query, make sure to itemize each field name, separated by commas, after the SELECT clause.

8. The WHERE clause is handy. It filters records so you see just those records you are interested in.
9. The WHERE clause in a SELECT statement is optional. If you omit the WHERE clause, you will get all available records in your record set.
10. Changing the WHERE clause affects the records you see in your query results.
11. Following the WHERE keyword is a logical expression. This logical expression is called a predicate.
12. Using the equal = sign finds exact criteria matches.
13. The T-SQL operator LIKE can be used to return a range of values. For example, **WHERE FirstName LIKE '[a-m]%'** returns records with a Firstname beginning with any letter between A and M (Ann through MaryAnn).
14. You can use wildcard characters in your WHERE clause.
15. The percent % symbol is the most common wildcard. This symbol represents any number of characters. For example, **WHERE Firstname like '%N'** would find a name that ends in N regardless of how long the name is. Examples may include Ann, MaryAnn, and Dean among others.
16. The % sign represent any number of characters, including zero. For example, **'%A%'** would find Lisa but would also find Alex.
17. If you want to “exact match” a % symbol, like the name R%per!est and all other names with a percent symbol in them, enclose the % symbol inside square brackets. For example, **LastName LIKE '%[%]%'**
18. If you miss a letter, forget a punctuation mark, or make a spelling error in your T-SQL code , SQL Server will return an error message.

## Joining Tables

Refer to Figure 1.17 for a look at Alex Adams and Barry Brown. Both these employees work in LocationID 1. If you were new and only had access to the Employee table, you would not have enough detailed information to send a package delivery to Alex Adams. What if we placed two tables next to one another? By physically drawing a line from the Employee.LocationID field to the Location.LocationID field we can get more location details for each employee. LocationID 1 is located at 111 First St. in Seattle, WA (see Figure 1.17).

Employee Table							
	EmplID	lastname	firstname	hiredate	LocationID	ManagerID	Status
1	1	Adams	Alex	2001-01-01 00:00:00.000	1	11	NULL
2	2	Brown	Barry	2002-08-12 00:00:00.000	1	11	NULL
3	3	Osako	Lee	1999-09-01 00:00:00.000	2	11	NULL
4	4	Kenison	David	1996-03-16 00:00:00.000	1		
5	5	Bender	Eric	2007-05-17 00:00:00.000	1		
6	6	Kendall	Lisa	2001-11-15 00:00:00.000	4		
7	7	Lonning	David	2000-01-01 00:00:00.000	1		
8	8	Marshbank	John	2001-11-15 00:00:00.000	NULL		
9	9	Newton	James	2003-09-30 00:00:00.000	2		
10	10	O'Haire	Terry	2004-10-04 00:00:00.000	2	3	NULL
11	11	Smith	Sally	1989-04-01 00:00:00.000	1	NULL	NULL
12	12	O'Neil	Barbara	1995-05-26 00:00:00.000	4	4	Has Tenure

Location Table				
	LocationID	street	city	state
1	1	111 First ST	Seattle	WA
2	2	222 Second AVE	Boston	MA
3	3	333 Third PL	Chicago	IL
4	4	444 Ruby ST	Spokane	WA

Figure 1.17 The Employee and Location tables are correlated on the LocationID field.

What about a global company with locations in all 50 states and over 100 different countries? We will have many records in our Location table and probably will not be able to look at both tables very efficiently on one screen.

## Inner Joins

Each query has only one result set and allows only one FROM clause. How can you include more than one table in a FROM clause? You can put up to 256 tables in one FROM clause if you use joins. You can even join a table to a copy of itself (Self Joins will be covered in Chapter 10). The most common type of join is called the *inner join*.

The inner join allows you to combine multiple tables in one query, but it requires a specific condition in order to do its work. You must ensure that the join statement has two tables with at least one common or overlapping field of a

similar datatype. The names do not have to match in order to join two tables (we will explore this further in Chapter 8). We already know the Employee and Location tables share a common LocationID field. The relationship is between Employee.LocationID and Location.LocationID, so just tell SQL Server that the join is ON this field and *voila!* You will combine two tables into one result set.

Every time a value is found in Employee.LocationID, the inner join searches for the matching record in the Location.LocationID field. If a match is found, data from both tables displays as a single record. Both tables will show all their fields if we type SELECT \* at the beginning of our query.

For visual purposes, let's put the Employee and Grant tables next to each other. We can see that the EmpID field correlates data between these two tables. Grant.EmpID equates to Employee.EmpID. If we look at EmpID 10 in both tables, we can see Terry O'Haire is the employee who found the \$41,000 grant

Employee Table					Grant Table				
	EmpID	lastname	firstname	hiredate	GrantID	GrantName	EmpID	Amount	
1	1	Adams	Alex	2001-01-01 00:00:00	1	92 Purr_Scents %% team	7	4750.00	
2	2	Brown	Barry	2002-08-12 00:00:00	2	K_Land fund trust	2	15750.00	
3	3	Osako	Lee	1999-09-01 00:00:00	3	Robert@BigStarBank.com	7	18100.00	
4	4	Kennson	David	1996-03-16 00:00:00	4	Norman's Outreach	NULL	21000.00	
5	5	Bender	Eric	2007-05-17 00:00:00	5	BIG 6's Foundation%	4	21000.00	
6	6	Kendall	Lisa	2001-11-15 00:00:00	6	TALTA_Kishan International	3	18100.00	
7	7	Lonning	David	2000-01-01 00:00:00	7	Ben@MoreTechnology.com	10	41000.00	
8	8	Marshbank	John	2001-11-15 00:00:00	8	@Last-U-Can-Help	7	25000.00	
9	9	Newton	James	2003-09-30 00:00:00	9	Thank you @.com	11	21500.00	
10	10	O'Haire	Terry	2004-10-04 00:00:00	10	Call Mom @Com	5	7500.00	
11	11	Smith	Sally	1989-04-01 00:00:00	1	NULL	NULL		
12	12	O'Neil	Barbara	1995-05-26 00:00:00	4	4	Has Tenure		

(Figure 1.18).

Again, placing these two small tables side by side and analyzing them can work, but it's time-consuming and not very efficient, especially if we have two very large tables. Instead, we can use SQL Server to combine both tables into one result set. Putting information into one report is of great value to businesses.

The tables in Figure 1.18 share the common field of EmpID. Knowing this is a key step.

To put all these records into one result set, you need both tables in the FROM clause. Joins require field(s) that correspond to both tables. The ON clause after the INNER JOIN shows this relationship (Figure 1.19).

If you looked closely at Figures 1.18 and 1.19 you may see a reason for caution. There are 10 grants, but the INNER JOIN only returned nine records. Thus, inner

**Figure 1.18** The Grant and Employee tables relate ON Grant.EmpID = Employee.EmpID.

joins can produce what seems to be a data loss. We will explore how to know when this will happen.

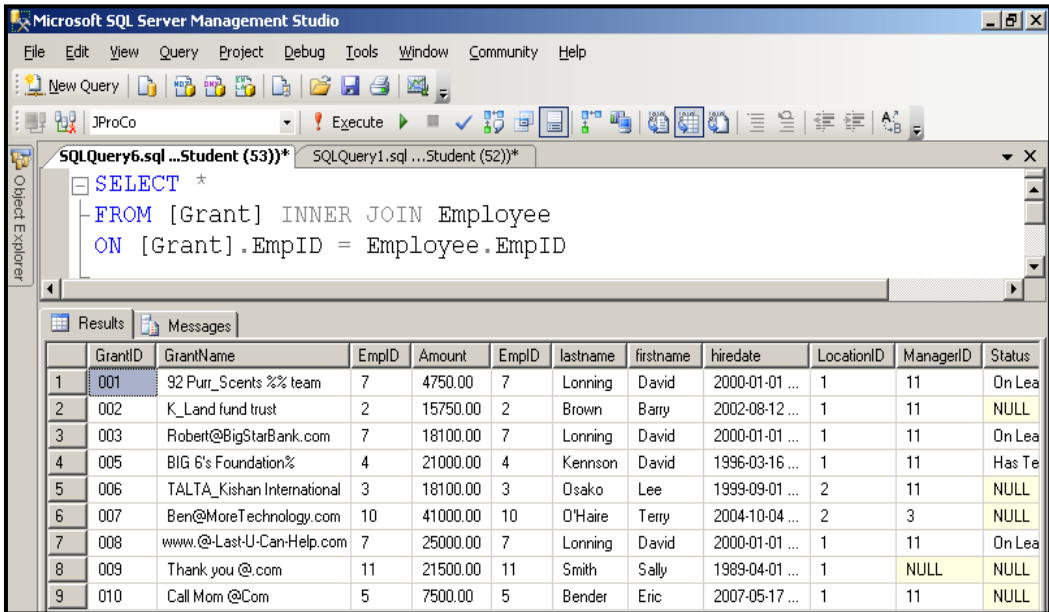


Figure 1.19 Joining the Grant and Employee tables using an inner join returns just nine records.

The Grant called “Norman’s Outreach” was an online registration, so no employee is listed as receiving credit. Since no match was found, the record containing the Norman’s Outreach grant was not included in the result set.

The core behavior of inner joins is to only include records when a match is found in both tables. Mismatched records are left out of the query result set.

## Outer Joins

Once you can handle inner joins, understanding outer joins is an easy progression. They both look for and display every match they find between two tables. Both joins require that you specify the matching field(s) in the ON clause. Outer joins can also show the mismatching records which inner joins omit.

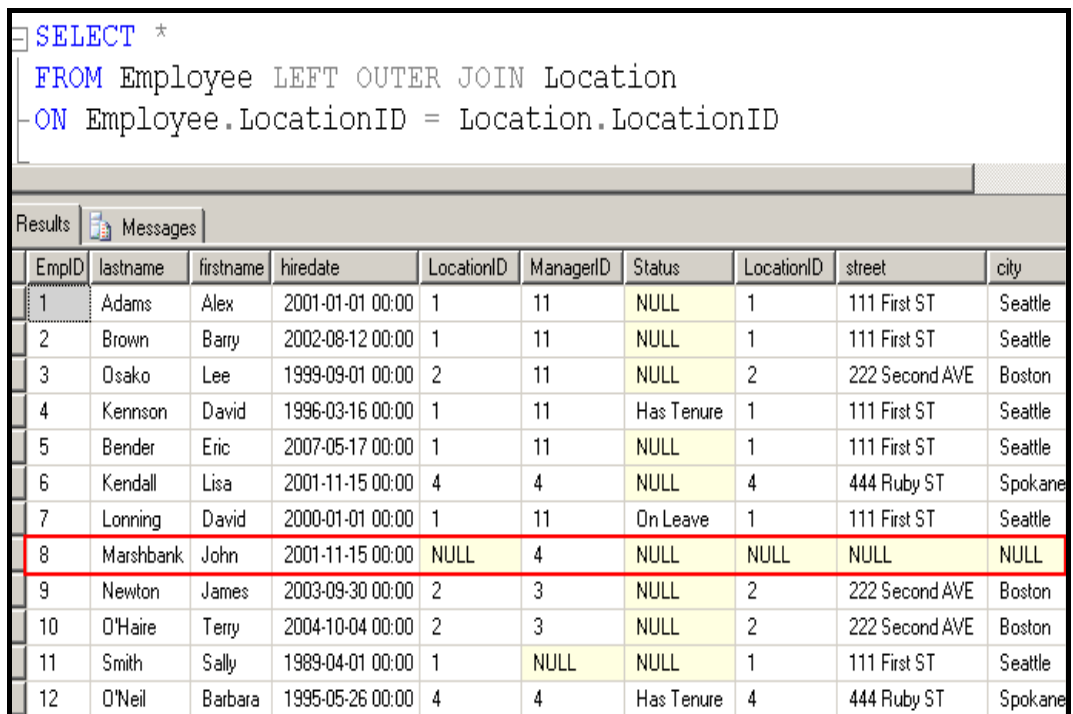
The Employee table has 12 records and the inner join query with the Location table shows only 11 records. That is because John Marshbank is JProCo’s traveling salesman and has no LocationID. In an employee report, you may want to list all employees and their location even if the employee(s) doesn’t have a location. To do this you need to use some type of outer join.

## Left Outer Joins

Before we fire up an outer join, we need to state our goal. There is an employee with no location. There is also a location under construction in Chicago that presently has no employees.

Are we looking for an employee report or a location report? In an employee report, it's OK if Chicago does not show in the results. We need to list all employees from the Employee table. Thus, the outer join must favor the Employee table.

In the query here, the Employee table is on the left. Since you want all employees included, you will make this a *left outer join* (Figure 1.20).



```
SELECT *
FROM Employee LEFT OUTER JOIN Location
ON Employee.LocationID = Location.LocationID
```

EmpID	lastname	firstname	hiredate	LocationID	ManagerID	Status	LocationID	street	city
1	Adams	Alex	2001-01-01 00:00	1	11	NULL	1	111 First ST	Seattle
2	Brown	Barry	2002-08-12 00:00	1	11	NULL	1	111 First ST	Seattle
3	Osako	Lee	1999-09-01 00:00	2	11	NULL	2	222 Second AVE	Boston
4	Kennson	David	1996-03-16 00:00	1	11	Has Tenure	1	111 First ST	Seattle
5	Bender	Eric	2007-05-17 00:00	1	11	NULL	1	111 First ST	Seattle
6	Kendall	Lisa	2001-11-15 00:00	4	4	NULL	4	444 Ruby ST	Spokane
7	Lonning	David	2000-01-01 00:00	1	11	On Leave	1	111 First ST	Seattle
8	Marshbank	John	2001-11-15 00:00	NULL	4	NULL	NULL	NULL	NULL
9	Newton	James	2003-09-30 00:00	2	3	NULL	2	222 Second AVE	Boston
10	O'Haire	Terry	2004-10-04 00:00	2	3	NULL	2	222 Second AVE	Boston
11	Smith	Sally	1989-04-01 00:00	1	NULL	NULL	1	111 First ST	Seattle
12	O'Neil	Barbara	1995-05-26 00:00	4	4	Has Tenure	4	444 Ruby ST	Spokane

**Figure 1.20** A left outer join favors the table listed to the *left of the join*, which is Employee.

Notice John Marshbank appears. This left outer join will find all records from the table on the left (Employee). It then tries to find matches with the table on the right (Location). All employees are shown and matches from Location are filled in and displayed.

## Right Outer Joins

If OSHA came to inspect all buildings, your goal would be to produce a list of all locations for the regulators. They want to see all locations and who works at those locations. They have no interest in John Marshbank, since he does not occupy a building. They want to see the buildings, even if they are under construction and no employees work there.

The screenshot shows the SQL Server Management Studio interface. The query window contains the following SQL code:

```
SELECT *
FROM Employee RIGHT OUTER JOIN Location
ON Employee.LocationID = Location.LocationID
```

The results window displays the following data:

EmpID	lastname	firstname	hiredate	LocationID	ManagerID	Status	LocationID	street	city
1	Adams	Alex	2001-01-01 00:00:00.000	1	11	NULL	1	111 First ST	Seattle
2	Brown	Barry	2002-08-12 00:00:00.000	1	11	NULL	1	111 First ST	Seattle
4	Kennson	David	1996-03-16 00:00:00.000	1	11	Has Tenure	1	111 First ST	Seattle
5	Bender	Eric	2007-05-17 00:00:00.000	1	11	NULL	1	111 First ST	Seattle
7	Lonning	David	2000-01-01 00:00:00.000	1	11	On Leave	1	111 First ST	Seattle
11	Smith	Sally	1989-04-01 00:00:00.000	1	NULL	NULL	1	111 First ST	Seattle
3	Osako	Lee	1999-09-01 00:00:00.000	2	11	NULL	2	222 Second AVE	Boston
9	Newton	James	2003-09-30 00:00:00.000	2	3	NULL	2	222 Second AVE	Boston
10	O'Haire	Terry	2004-10-04 00:00:00.000	2	3	NULL	2	222 Second AVE	Boston
NULL	NULL	NULL	NULL	NULL	NULL	NULL	3	333 Third PL	Chicago
6	Kendall	Lisa	2001-11-15 00:00:00.000	4	4	NULL	4	444 Ruby ST	Spokane
12	O'Neil	Barbara	1995-05-26 00:00:00.000	4	4	Has Tenure	4	444 Ruby ST	Spokane

**Figure 1.21** The right outer join shows all locations, even if nobody works there.

To show all records from the table on the right, you need a *right outer join* (Figure 1.21). Since the right outer join favors the table listed to the right of the join (i.e., the Location table), all locations will be displayed. The employee list for Chicago is NULL. However, Chicago still appears as a location.

Notice the EmpID values are no longer in order. The LocationID values go from one to four in order. The Location table is showing all records in order and any matching values it finds in the join. Location is on the right side of this *right outer join*.

## Full Outer Joins

So far we have seen a left outer join works great when you want to see all employees, even if they have no location. What about seeing all locations, even without employees? The right outer join is just the thing for a location report showing all locations, even ones where there are no employees.

How about if you wanted to see all employees and all locations regardless of whether each employee matches a location and regardless of whether each location matches to an employee? This is a *full outer join* (Figure 1.22).

The screenshot shows the SQL Server Management Studio interface. The query editor contains the following SQL code:

```
SELECT *
FROM Employee FULL OUTER JOIN Location
ON Employee.LocationID = Location.LocationID
```

The Results pane displays the following data:

EmpID	lastname	firstname	hiredate	LocationID	ManagerID	Status	LocationID	street	city	state
1	Adams	Alex	2001-01-01 ...	1	11	NULL	1	111 First ST	Seattle	WA
2	Brown	Barry	2002-08-12 ...	1	11	NULL	1	111 First ST	Seattle	WA
3	Osako	Lee	1999-09-01 ...	2	11	NULL	2	222 Second AVE	Boston	MA
4	Kennson	David	1996-03-16 ...	1	11	Has Tenure	1	111 First ST	Seattle	WA
5	Bender	Eric	2007-05-17 ...	1	11	NULL	1	111 First ST	Seattle	WA
6	Kendall	Lisa	2001-11-15 ...	4	4	NULL	4	444 Ruby ST	Spokane	WA
7	Lonning	David	2000-01-01 ...	1	11	On Leave	1	111 First ST	Seattle	WA
8	Marshbank	John	2001-11-15 ...	NULL	4	NULL	NULL	NULL	NULL	NULL
9	Newton	James	2003-09-30 ...	2	3	NULL	2	222 Second AVE	Boston	MA
10	O'Haire	Terry	2004-10-04 ...	2	3	NULL	2	222 Second AVE	Boston	MA
11	Smith	Sally	1989-04-01 ...	1	NULL	NULL	1	111 First ST	Seattle	WA
12	O'Neil	Barbara	1995-05-26 ...	4	4	Has Tenure	4	444 Ruby ST	Spokane	WA
NULL	NULL	NULL	NULL	NULL	NULL	NULL	3	333 Third PL	Chicago	IL

**Figure 1.22** A full outer join shows all records from both tables, including matches & mismatches.

Notice with a full outer join you can get more records than either table contains. Here, the result set has thirteen records. The Employee table has twelve records and the Location table has four. Full outer joins show all matches and mismatches from both tables.

Here is another example of the full outer join with mismatching records from both tables. You want to find all employees with no grants and all grants with no employees. Use the full outer join seen in Figure 1.23.

```
SELECT *
FROM [Grant] FULL OUTER JOIN [Employee]
ON [Grant].EmpID = [Employee].EmpID
```

GrantID	GrantName	EmpID	Amount	EmpID	lastname	firstname	hiredate	LocationID	ManagerID	Status
001	92 Purr_Scents %% team	007	4750.00	007	Lonning	David	2000-01-01 00:00:00.000	001	011	On Leave
002	K_Land fund trust	002	15750.00	002	Brown	Bary	2002-08-12 00:00:00.000	001	011	NULL
003	Robert@BigStarBank.com	007	18100.00	007	Lonning	David	2000-01-01 00:00:00.000	001	011	On Leave
004	Norman's Outreach	NULL	5500.00	NULL	NULL	NULL	NULL	NULL	NULL	NULL
005	BIG 6's Foundation%	004	21000.00	004	Kennison	David	1996-03-16 00:00:00.000	001	011	Has Tenure
006	TALTA_Kishan International	003	18100.00	003	Osako	Lee	1999-09-01 00:00:00.000	002	011	NULL
007	Ben@MoreTechnology.com	010	41000.00	010	O'Haire	Terry	2004-10-04 00:00:00.000	002	003	NULL
008	@Last-U-Can-Help	007	25000.00	007	Lonning	David	2000-01-01 00:00:00.000	001	011	On Leave
009	Thank you @.com	011	21500.00	011	Smith	Sally	1989-04-01 00:00:00.000	001	NULL	NULL
010	Call Mom @Com	005	7500.00	005	Bender	Eric	2007-05-17 00:00:00.000	001	011	NULL
NULL	NULL	NULL	NULL	001	Adams	Alex	2001-01-01 00:00:00.000	001	011	NULL
NULL	NULL	NULL	NULL	006	Kendall	Lisa	2001-11-15 00:00:00.000	004	004	NULL
NULL	NULL	NULL	NULL	008	Marshbank	John	2001-11-15 00:00:00.000	NULL	004	NULL
NULL	NULL	NULL	NULL	009	Newton	James	2003-09-30 00:00:00.000	002	003	NULL
NULL	NULL	NULL	NULL	012	O'Neil	Barbara	1995-05-26 00:00:00.000	004	004	Has Tenure

Figure 1.23 A full outer join between the Grant and Employee tables shows all records.

As we can see, the full outer join finds all matches and mismatches. All our joins so far have not used the WHERE clause, so we got all possible records. With a WHERE clause you would get only the records that satisfied the WHERE criteria. For example, we could add the clause WHERE LocationID = 1 to the code shown above (Figure 1.23). This WHERE clause would filter the records down to just the Seattle location and the join matches and mismatches that remain will display. Without the WHERE clause, you get all matches and mismatches in your result set. Mismatches between the tables show up as NULL.

## Table Aliasing

*Table aliasing* is a big saver of keystrokes. As we know, tables are listed in the FROM clause. We had to retype the table names again in the ON clause as seen in the code below:

```
SELECT *
FROM Location INNER JOIN Employee
ON Location.LocationID = Employee.LocationID
```

## Setting Aliases

Our next goal is to get the FROM clause to refer to the Employee table as “Emp” and the Location table as “Loc.” It’s like giving your friend a nickname. They respond and use it, but their birth certificate still has the original name. In other words, the tables will not change names literally, but your query can use a shorter name. Simply rename the table “AS *alias*” in the FROM clause and you then use the shorter name. Here is a T-SQL example:

```
SELECT *  
FROM Location AS Loc INNER JOIN Employee AS Emp  
ON Loc.LocationID = Emp.LocationID
```

In this example, the payoff is only slightly obvious. Some queries will use the table name in dozens of places. It’s far quicker to type “Loc” many times versus “Location.”

## Using Aliases

At this point you are probably not a big fan of two-part field names in a query. In fact, the simple name does the same thing with far fewer keystrokes. In comparing the two queries below, we see identical results.

### Simple Field Names

```
SELECT FirstName, LastName, [State]  
FROM Location INNER JOIN Employee  
ON Location.LocationID = Employee.LocationID
```

### Two-part Field Names

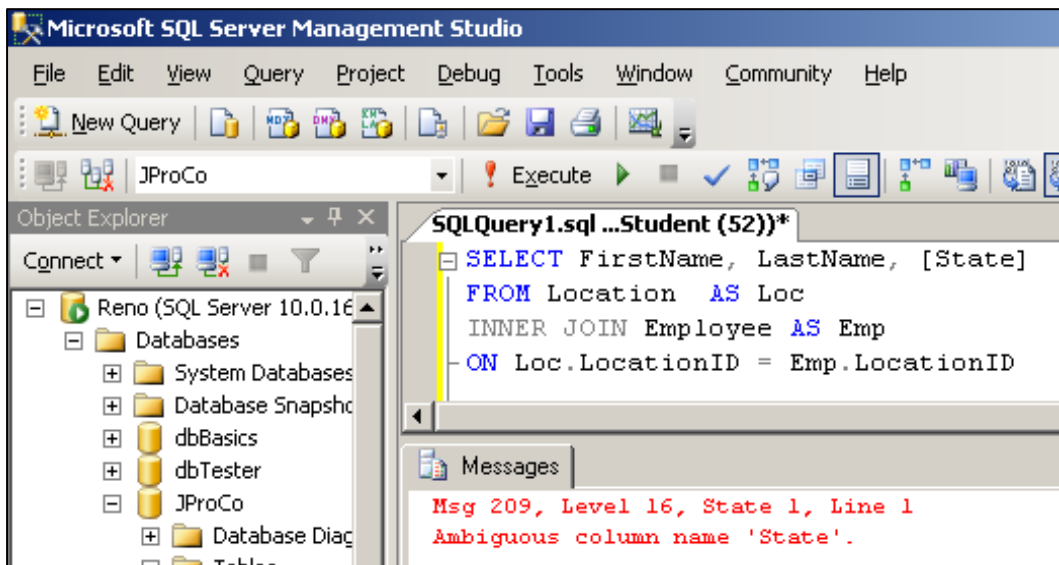
```
SELECT Employee.FirstName, Employee.LastName,  
Location.[State]  
FROM Location INNER JOIN Employee  
ON Location.LocationID = Employee.LocationID
```

The truth is we lucked out on all our past queries. The simple field name query from above is looking for FirstName from either table. There is a FirstName field in Employee and none in Location. Thus, FirstName here actually means Employee.FirstName, so SQL Server implicitly does this behind the scenes for you. The same is true with the [State] field. When SQL Server looks for the [State] field in the Employee table and realizes it isn’t present, it does us the courtesy of pulling it in from the Location table.

This brings up a question. If both tables had a State field, would SQL Server choose the Employee.State field or the Location.State field? If SQL Server detects any ambiguity, it will give an error message and request more information.

However, if you ran a SELECT \* statement in that same scenario (i.e., both tables have a field named State), you would get both fields in your result and no error message. For this reason, a best practice we emphasize in this book is to always build your queries using SELECT \* up until the moment your joins are all working well and you need to list the precise fields you will include in your report.

When you itemize fields, you must specify exactly which one you want. Suppose you would like to include the State field in your report. In our imaginary scenario, if you used just the simple name “State” when multiple fields with the same name exist, your specification will be ambiguous and SQL Server will display an error.

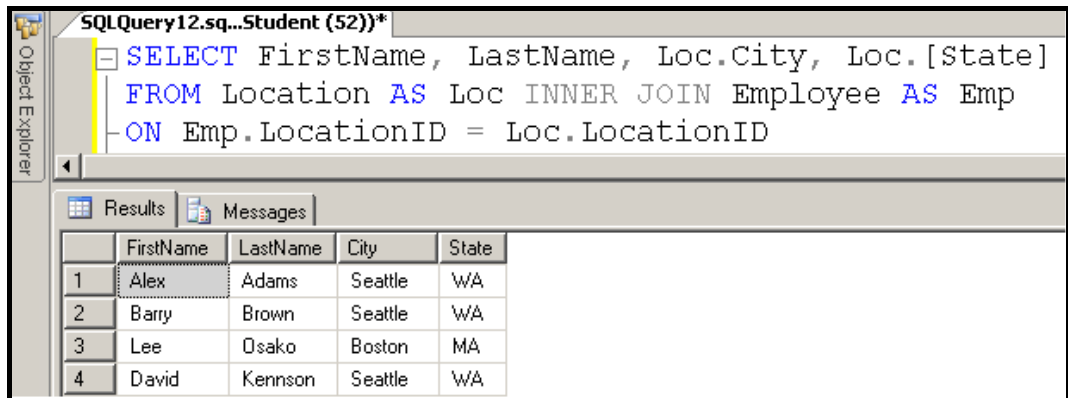


**Figure 1.24** In our scenario, SQL Server gives an error since it can't tell which field to choose.

Notice the error message is coupled with an explanation that SQL Server cannot guess what information you want to see. The only time it allows simple field names in a multiple table query is when there exists only field with that name. This would be exactly the type of situation where we want to use SELECT \* instead of an itemized field list, so that we don't risk ambiguous column name errors while we are trying to build our joins.

Now let's specify that it's the State field from the Location table that we want. We specify that by using the the two-part field name. In your field selection list,

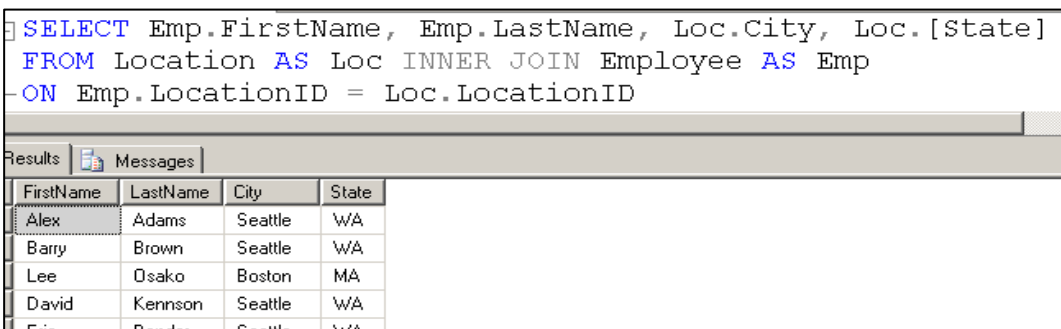
specify the Location.State field with the table alias Loc (Loc.State). Let’s also put an alias on the field City (Fig 1.25).



**Figure 1.25** Using the alias Loc for the Location table to create two-part names for City and State.

In our example, we had to prefix State with the Loc alias. We did not need to prefix the City field in order to get this query to work. But consider this – can you guarantee that tomorrow someone will not add a City field to the Employee table to denote where the employee resides? If this happened, some of your queries would break because SQL Server would find multiple fields named “City” in the query and not know which one to choose.

In your SQL career, you will often write queries you need to work today and into the future. A good way to do this is to use two-part names for all fields listed in your SELECT statement. It’s a little extra typing, but aliasing your tables significantly reduces keystrokes. The shorter prefixes also make your code more readable. Note in Figure 1.26 we have changed every listed field as a two-part name. The aliasing we did in our FROM clause allowed us to use the shorter names.



**Figure 1.26** Aliasing is a time-saving way to create more robust and durable code.

## Lab 1.2: Joining Tables

**Lab Prep:** Before you can begin the lab you must run the SQLQueriesChapter1.2Setup.sql script.

**Skill Check 1:** In one query you want to see the employees and cities where they work. You know the two tables share a common LocationID field. Query both of these JProCo tables by joining them on this field.

Your field select list should only include FirstName, LastName, City, and State. Alias the Employee table as “em” and the Location table as “lo.” Use two-part names in all areas of your query. When you are done, your result should resemble Figure 1.27.

FirstName	LastName	City	State
Alex	Adams	Seattle	WA
Barry	Brown	Seattle	WA
Lee	Osako	Boston	MA
David	Kennson	Seattle	WA
Eric	Bender	Seattle	WA
Lisa	Kendall	Spokane	WA
David	Lonning	Seattle	WA
James	Newton	Boston	MA
Terry	O'Haire	Boston	MA
Sally	Smith	Seattle	WA
Barbara	O'Neil	Spokane	WA

Figure 1.27 Result contains 11 records.

**Skill Check 2:** Open a query window to the JProCo database. Write a query that shows the list of all grants and the first and last names of the employees who acquired them. If the grant was not found by an employee, we still want to see a NULL first and last name.

FirstName	LastName	GrantName	Amount
David	Lonning	92 Purr_Scents %% team	4750.00
Barry	Brown	K_Land fund trust	15750.00
David	Lonning	Robert@BigStarBank.com	18100.00
NULL	NULL	Norman's Outreach	21000.00
David	Kennson	BIG 6's Foundation%	21000.00
Lee	Osako	TALTA_Kishan International	18100.00
Terry	O'Haire	Ben@MoreTechnology.com	41000.00
David	Lonning	www.@-Last-U-Can-Help.com	25000.00
Sally	Smith	Thank you @.com	21500.00
Eric	Bender	Call Mom @Com	7500.00

Figure 1.28 Skill Check 2 shows 10 records.

To do this, join the [Employee] and [Grant] tables. Show the FirstName, LastName, GrantName and Amount fields. Alias the Employee table as em and the [Grant] table as gr. Use two part names in all areas of your query. When you are done, your result should resemble Figure 1.28.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab1.2\_JoiningTables.sql.

## Outer Joins - Points to Ponder

1. Tables with related fields can be used together in a single query using a join.
2. An inner join only returns a result set with perfectly matched values from fields in two or more tables.
3. An inner join is the default join type. If the term “inner” is omitted from the join clause of a query, SQL Server will assume it to be an inner join.
4. Outer joins can allow more records to be seen in your result set than just an equal record match list from an inner join.
5. There are three types of outer joins: left outer join, right outer join and full outer join.
6. In a left outer join, the table named first will have all its records appear, even if SQL Server finds no matching records from the table listed second (i.e., appearing to the right of the LEFT OUTER JOIN clause).
7. In a right outer join, the table listed second might have records that appear even if no matching records are found in the table you listed first.
8. In full outer joins, all matches and mismatches are displayed from both tables.
9. Using the word OUTER is highly recommended but is optional. LEFT OUTER JOIN means the same thing as LEFT JOIN in a query.
10. When you alias a table, you use an abbreviation. T-SQL aliasing usually means using a shorter name than the original identifier.
11. Aliases must be declared immediately after the table’s name in the FROM clause.
12. When you alias a table, you must use the alias wherever you refer to that table in your query. This process makes your queries unambiguous.
13. Using the keyword AS when specifying an alias is optional. It is recommended because it makes your code easier to read but is often omitted.

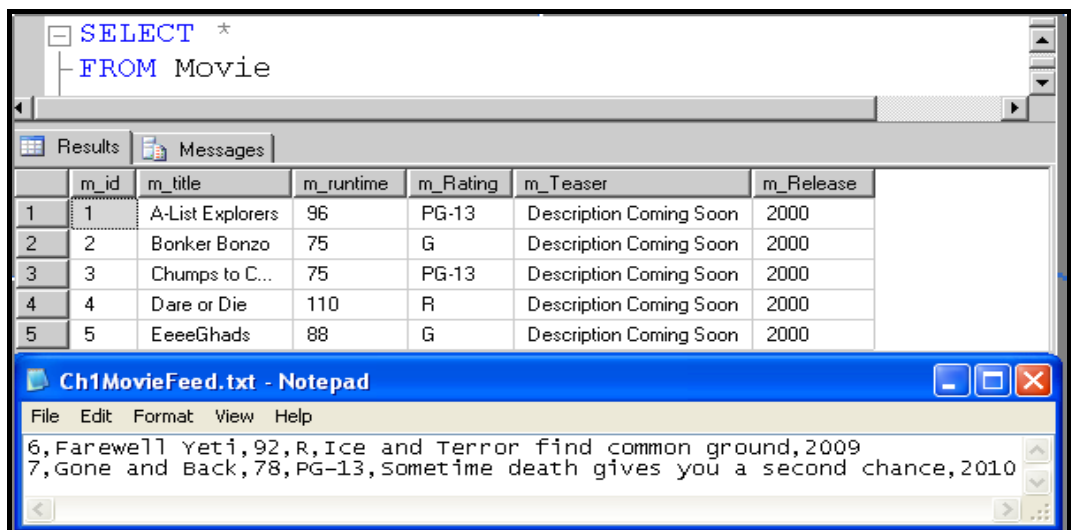
## Bulk Copy Program (BCP)

There are utilities that understand data, and SQL Server can manage the inserting of data for you. Some programs like *SQL Server Integration Services* (SSIS) can take just about any source of data and pump it into, or out of, SQL Server. The *Bulk Copy Program* (BCP) imports/exports data into/out of SQL. However, the BCP output from SQL is always a file. In this chapter, we explore how to use BCP, which was designed for the flow of data between SQL Server and text files. It's a simple utility that does one of the most common types of bulk copying.

### Importing with BCP

If you have data in a text file and you need to move it into a table, you're ready to import that file into SQL Server using BCP. A quick visual comparison of the data in the input file and the destination table will really help. We need to confirm that the text file can supply enough data in the correct format to populate each field for each record in the destination table.

In the database dbMovie, we have five records in our Movie table and two records in the file Ch1MovieFeed.txt. We want to import movies 6 and 7 into the Movie table (see Figure 1.29). Locate the file Ch1MovieFeed.txt in the Resources folder of your companion CD. Copy this file into your C:\Joes2Pros folder.



**Figure 1.29** A text file with the right amount of data to make 2 records will be copied into Movie.

The first record of the text file is `m_id = 6`. Note the value 6 is terminated with a comma to separate it from the second field value of "Farewell Yeti." After the next separating comma is the third field, which shows a runtime of 92 minutes. The pattern repeats until the final field in each record is reached. At that point, a

return (a carriage return or hitting the Enter key) signals the end of the record. Just remember that *commas separate fields*, and after each line the *return separates records*.

OK, we've confirmed the data in the source file and now are confident the data conforms to the destination format (the Movie table). BCP should run without a problem. We first need to provide a few logistical items to BCP: the filepath of the source file, which table we want the data copied into, and some other basics, such as the designation of commas as *field terminators*. Close out of the Movie database. From your desktop click Start > Run. Type in the letters CMD in the Run box and hit OK (Figure 1.30).

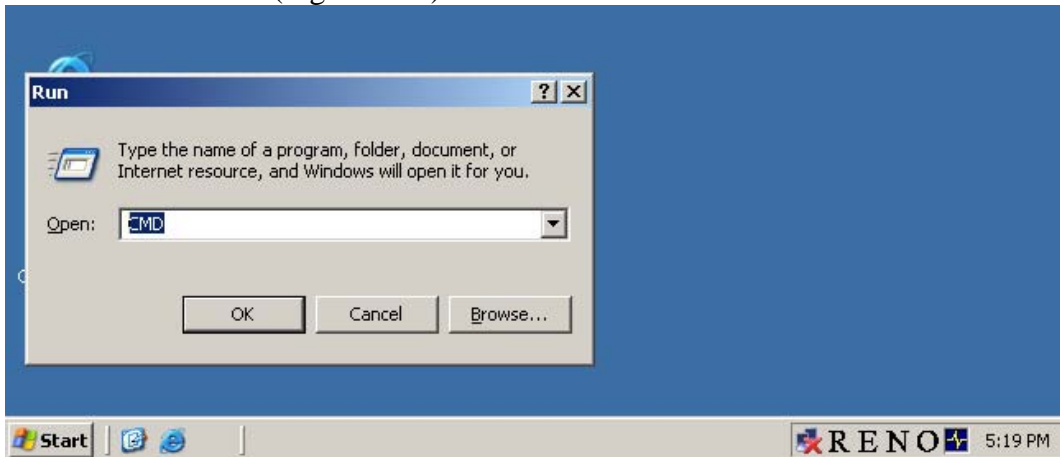


Figure 1.30 Starting the command prompt.

With the command prompt open (see Figure 1.30), we can go to the root of the C: drive with a “cd\” command and hit Enter. Go into the folder with a “cd Joes2Pros” command.

While in the folder, a good practice is to make sure the Ch1MovieFeed.txt is in the directory. You can type “dir” to see this listing. Once you know it’s there, you can proceed to invoke BCP.

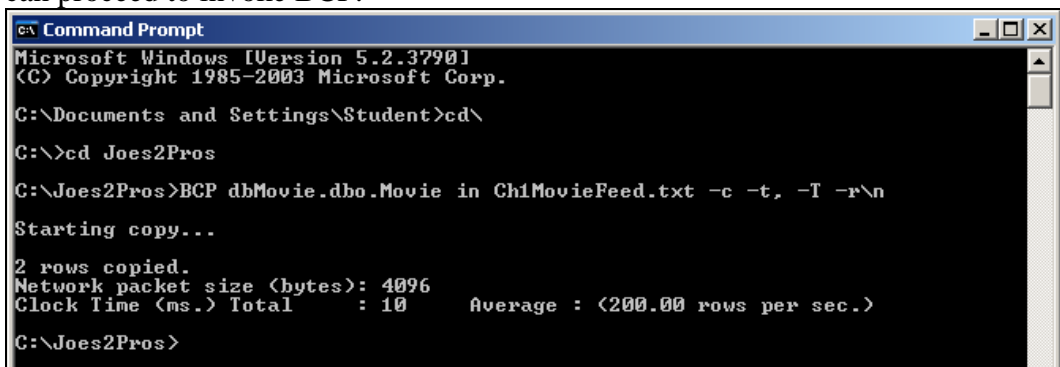


Figure 1.31 After pathing to C:\Joes2Pros folder, we ran BCP to pump 2 new records into Movie.

We want to tell BCP to expect character data (-c) with fields terminated by commas (-t). Windows has already authenticated our password. SQL trusts Windows. Since we are logged on as a user with permissions, we don't want to re-type our password. Use the Trusted connection with an upper case (-T). Lastly, we have multiple records in the text file. Each record is separated by a new line (\n) in your text file. Put all these command switches together as you see in Figure 1.31.

**BCP dbMovie.dbo.Movie in Ch1MovieFeed.txt -c -t, -T -r\n**

Query your Movie table and you will notice two new records for a total of seven (see Figure 1.32). The m\_id 6 and 7 were successfully added using BCP!

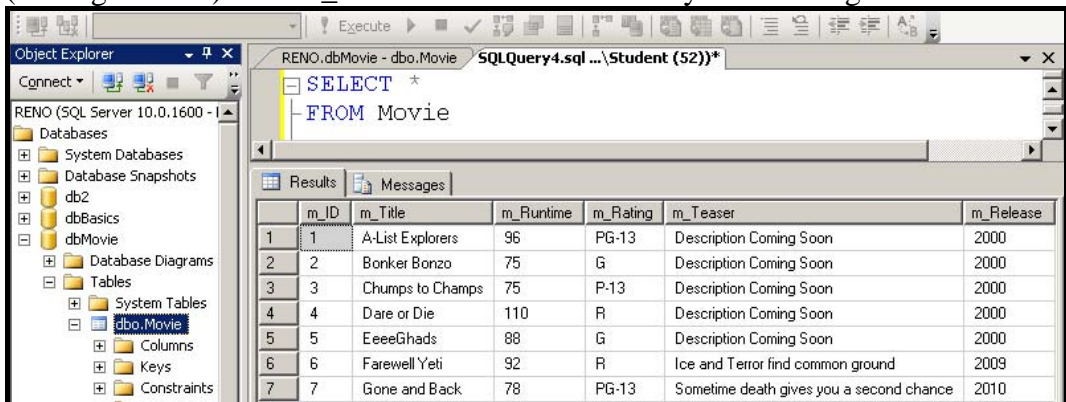


Figure 1.32 After running BCP, the result set shows there are now seven records.

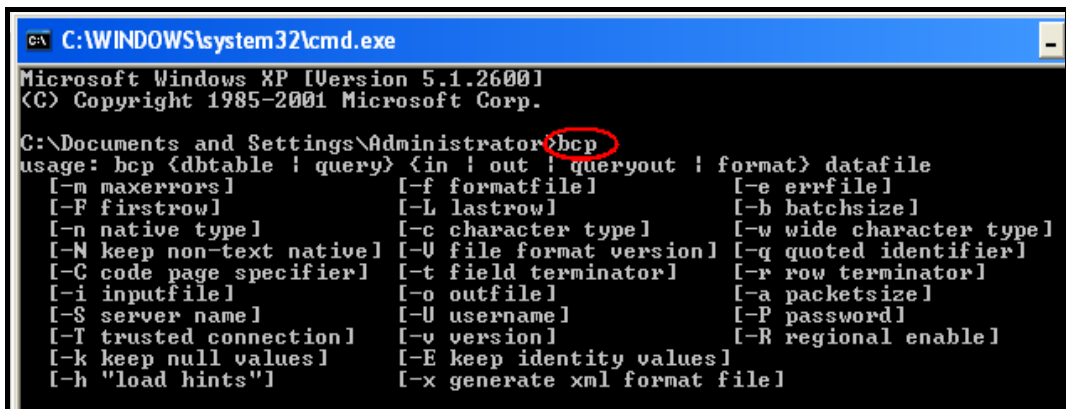


Figure 1.33 To see the BCP command menu, type “BCP” from the command line.

Type “bcp” at the command line. (Start > Run > CMD > BCP) to see a list of the BCP commands (see Figure 1.33). For a review of exporting files (BCP Out) from your SQL Server database, see *Beginning SQL Joes 2 Pros* (Chapter 7 “Exporting Data”).

## Lab 1.3: Using BCP

**Lab Prep:** Before you can begin the lab you must run the SQLQueriesChapter1.3Setup.sql script.

**Skill Check 1:** Copy the Ch1SalesInvoiceFeed.txt file to your C:\Joes2Pros folder. Run BCP with the right switches to put 1877 records into the JProCo.dbo.SalesInvoice table as seen in Figure 1.34.

```
Starting copy...
1000 rows sent to SQL Server. Total sent: 1000

1877 rows copied.
Network packet size (bytes): 4096
Clock Time (ms.) Total      : 121      Average : (15512.40 rows per sec.)

C:\Joes2Pros>
```

Figure 1.34 BCP has copied 1877 rows into your SalesInvoice table

**Skill Check 2:** Copy the Ch1CustomerFeed.txt file to your C:\Joes2Pros folder. Run BCP with the right switches to populate the Customer table in the JProCo database. When you're done, you should see the 770 rows added to your existing 5 customers. You will have a total of 775 records in your Customer table. To verify your result, run a query of all records from the SalesInvoice table and see that you have 1877 customers. Your result should resemble the figure you see here (Figure 1.35).

	InvoiceID	OrderDate	PaidDate	CustomerID	Comment
1	1	2006-01-03 00:00:00.000	2006-01-11 03:22:44.587	472	NULL
2	2	2006-01-04 02:22:41.473	2006-02-01 04:15:34.590	388	NULL
3	3	2006-01-04 05:33:01.150	2006-02-14 13:45:02.580	279	NULL
4	4	2006-01-04 22:06:58.657	2006-02-08 22:06:14.247	309	NULL
5	5	2006-01-05 11:37:45.597	2006-02-10 20:01:26.540	757	NULL
6	6	2006-01-06 23:53:14.320	2006-01-28 22:48:05.997	493	NULL
7	7	2006-01-08 08:06:33.210	2006-02-05 08:41:58.453	209	NULL

Query executed successfully. RENO (10.0 RTM) RENO\Administrator (53) JProCo 00:00:00 1877 rows

Figure 1.35 The SalesInvoice table is populated.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab1.3\_BCP.sql.

## Using BCP - Points to Ponder

1. BCP stands for Bulk Copy Program.
2. BCP lets you to perform data imports and exports using a command-line utility.
3. In BCP, the `-t` switch is used to specify how your fields are terminated. For example, if you use commas between each field, use `-t,` and if you use ampersands between them, use `-t&` for the switch.
4. Upper and lower case switches have different meanings in BCP and all Command Prompt utilities.
5. To see the list of BCP usage commands, type “BCP” from the command line. Start > Run > CMD > BCP

## Creating Tables

To build a new table you need the CREATE TABLE statement. CREATE is a DDL statement (If you need a review, then see *Beginning SQL Joes 2 Pros* Chapter 5 “Data Definition Language” and Chapter 11 “SQL Language Statement Types”).

**Do not run this code yet** – later you will run it as part of Lab 1.4.

```
USE JProCo
GO
CREATE TABLE SalesInvoiceDetail
(InvoiceDetailID int PRIMARY KEY,
 InvoiceID int NOT NULL,
 ProductID int NOT NULL,
 Quantity int NOT NULL,
 UnitDiscount smallmoney NULL)
GO
```

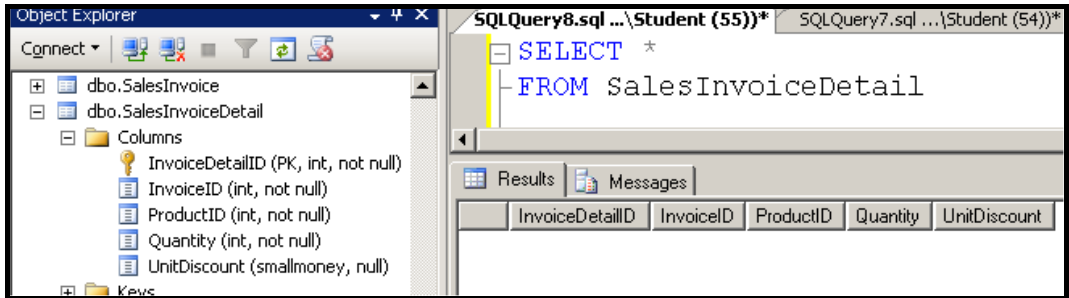
**Figure 1.36** This is the T-SQL code used to create the SalesInvoiceDetail table.

This T-SQL code creates the SalesInvoiceDetail table. The field names and data types are in the parentheses. The InvoiceDetailID is the primary key for the table, which means you can't have two records with the same InvoiceDetailID. Fields with Primary keys do not allow nulls even if you don't specify NOT NULL when creating the field. (*Note: Primary key details are outside of the scope of this section and will be covered more in Book 3.*) The second field definition states that the InvoiceID cannot contain null values. The ProductID field will accept integer (int) data and cannot be null. If the UnitDiscount is not known, you can leave that null until a later time. You can create records and still enter some of the values at a later date when they are known. (This topic will be further explored in Chapter 3.)

The following is a preview of the steps you will take in Lab 1.4 to create, populate, and query a table. The screenshots are a visual recap of the steps you would take to create a new table and populate it with records.

### Step 1. CREATE TABLE

SQL Server builds the table structure with the CREATE statement shown in Figure 1.36. Query the table with a SELECT clause as shown in Figure 1.37. Since this table is brand new, it has no records. The unpopulated table is seen in Figure 1.37 with the five fields you created.

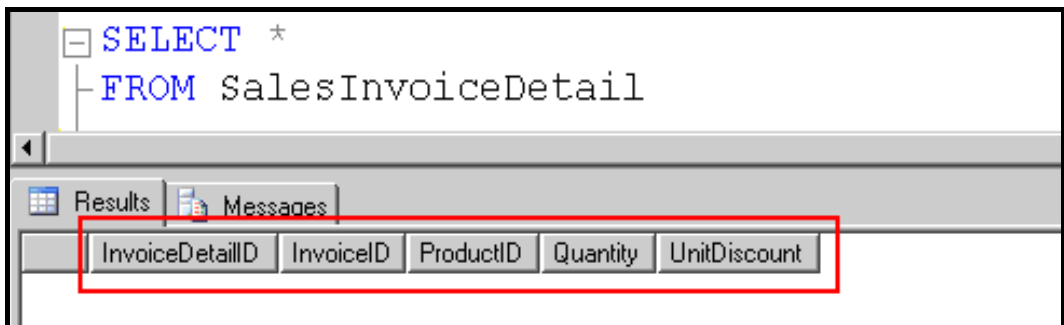


**Figure 1.37** This query shows the SalesInvoiceDetail table, which is not yet populated with data.

## Inserting Data

Tables are created for the purpose of holding data. Every table begins its life as an unpopulated table. The only way to populate a table is by inserting data. T-SQL coding that starts with the INSERT keyword are DML statements. INSERT statements add new records to tables.

Currently the SalesInvoiceDetail table is unpopulated, meaning it has no records. How much data you need to add depends on how many fields the table has. A quick look at the SalesInvoiceDetail table shows we have five fields (Fig 1.38).



**Figure 1.38** The SalesInvoiceDetail table has five fields.

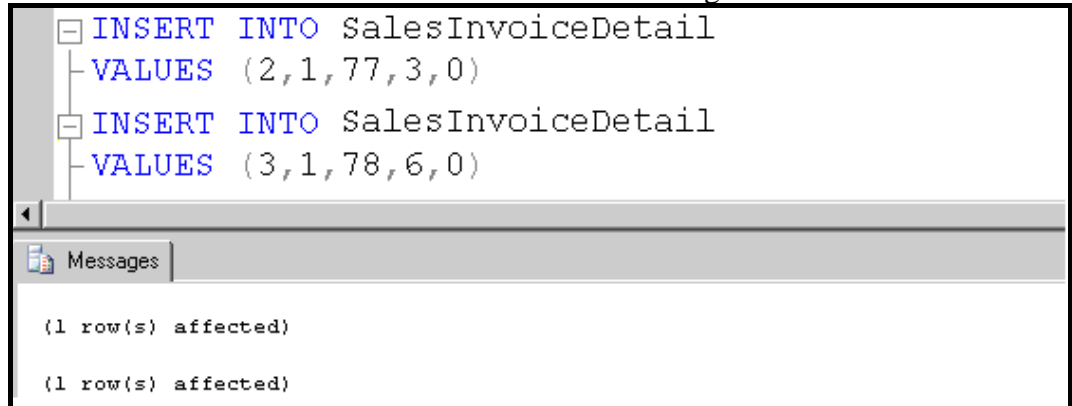
### Step 2. INSERT INTO *Table* VALUES (insert the first record).

To populate the first record, we need to supply five values. In an INSERT statement you separate each field value with a comma. To add the first record, use the following code:

```
INSERT INTO SalesInvoiceDetail VALUES  
(1,1,76,2,0)
```

**Step 3. INSERT INTO Table VALUES (insert the next two records).**

If you want to insert two records at the same time, you always have the option to run multiple INSERT INTO statements together. We can insert InvoiceDetailID 2 and InvoiceDetailID 3 at the same time as shown in Figure 1.39.

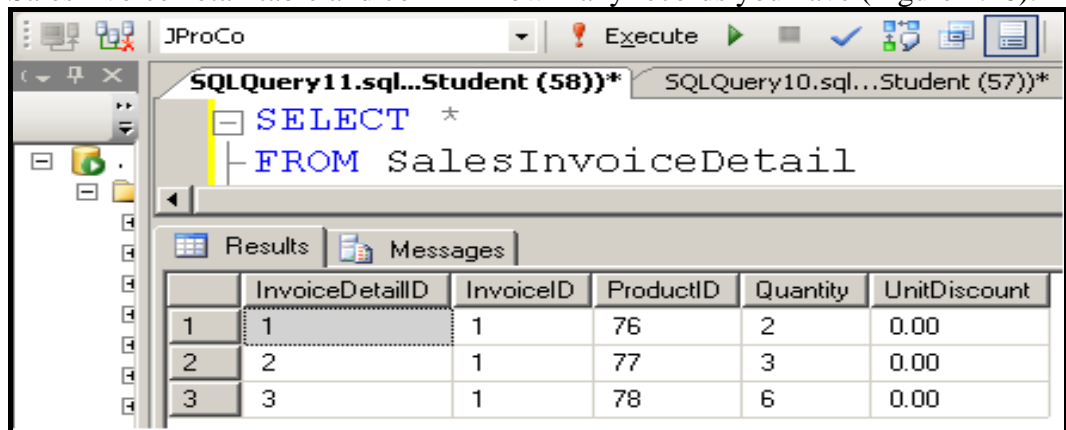


**Figure 1.39** Two INSERT INTO statements each inserted one record.

There was no need to run these INSERT INTO statements in separate batches with GO statements. DML statements like INSERT INTO use Transaction Control Language (TCL) instead of batches. Each insert statement in Figure 1.39 inserted one record, resulting in two additional records in the SalesInvoiceDetail table (see Figure 1.39). *Note that this example shows the SQL 2005 INSERT syntax where an insert statement can only insert one record. In 2008 you can insert many records at once with one insert statement. All remaining INSERT statements in this book will use the SQL Server 2008 row constructor syntax.*

**Step 4. SELECT \* FROM SalesInvoiceDetail (query the table).**

Run a simple SELECT \* FROM query to show all records from the SalesInvoiceDetail table and confirm how many records you have (Figure 1.40).



**Figure 1.40** The table SalesInvoiceDetail now contains three records.

## Lab 1.4: Creating and Populating Tables

**Lab Prep:** Before you can begin the lab you must run the SQLQueriesChapter1.4Setup.sql script.

**Skill Check 1:** Create the dbo.StateList table in the JProCo database with the design you see in Figure 1.41. Note: This table has no Primary key.

Column Name	Data Type	Allow Nulls
StateID	char(2)	<input type="checkbox"/>
StateName	varchar(50)	<input type="checkbox"/>
RegionName	varchar(50)	<input checked="" type="checkbox"/>
LandMass	int	<input checked="" type="checkbox"/>

**Figure 1.41** The design view of the StateList table.

Next you will use BCP to import Ch1StateListFeed.txt into the StateList table you created. Then check your result by running a SELECT from the table. Your result and record count should match the figure below (see Figure 1.42).

StateID	StateName	RegionName	LandMass	
1	AK	Alaska	USA	656425
2	AL	Alabama	USA-Continental	52423
3	AR	Arkansas	USA-Continental	53182
4	AZ	Arizona	USA-Continental	114006
5	CA	California	USA-Continental	163707
6	CO	Colorado	USA-Continental	104100
7	CT	Connecticut	USA-Continental	5544

**Figure 1.42** The StateList table after you BCP in the 53 records from the Ch1StateListFeed.txt file.

**Skill Check 2:** Drop, re-create, then populate the SalesInvoiceDetail table in the JProCo database. Use the design you see in Figure 1.43.

RENO.JProCo - ...sInvoiceDetail*			
	Column Name	Data Type	Allow Nulls
🔑	InvoiceDetailID	int	<input type="checkbox"/>
	InvoiceID	int	<input type="checkbox"/>
	ProductID	int	<input type="checkbox"/>
	Quantity	int	<input type="checkbox"/>
	UnitDiscount	smallmoney	<input checked="" type="checkbox"/>

**Figure 1.43** The design view of SalesInvoiceDetail

Populate the table with the Ch1SalesInvoiceDetailFeed.txt using BCP. During your import into the database, your result will look like Figure 1.44 below.

```
Starting copy...
1000 rows sent to SQL Server. Total sent: 1000
1000 rows sent to SQL Server. Total sent: 2000
1000 rows sent to SQL Server. Total sent: 3000
1000 rows sent to SQL Server. Total sent: 4000
1000 rows sent to SQL Server. Total sent: 5000
1000 rows sent to SQL Server. Total sent: 6000

6960 rows copied.
Network packet size (bytes): 4096
Clock Time (ms.) Total      : 341      Average : (20410.56 rows per sec.)

C:\joes2pros>
```

**Figure 1.44** BCP copied 6960 records into the SaleInvoiceDetail table.

A quick check of your table will show the records you see here in Figure 1.45.

	InvoiceDetailID	InvoiceID	ProductID	Quantity	UnitDiscount
1	1	1	76	2	0.00
2	2	1	77	3	0.00
3	3	1	78	6	0.00
4	4	1	71	5	0.00

Query executed successfully (local) (10.0 RTM) RENO\Student (52) JProCo 00:00:00 6960 rows

**Figure 1.45** A simple query of SalesInvoiceDetail shows the table is now populated.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab1.4\_CreatingAndPopulatingTables.sql

## Creating and Populating Tables-Points to Ponder

- 1.** The `CREATE TABLE` statement is the T-SQL way of adding a table object to a database.
- 2.** A `CREATE TABLE` statement is a DDL (Data Definition Language) statement.
- 3.** `CREATE` is used to make new objects in SQL Server including tables, or even whole databases. The `CREATE` clause is part of a DDL statement.
- 4.** `DROP` removes objects in SQL Server (tables, databases). The `DROP` keyword is the beginning of a DDL statement.
- 5.** At the time that you create a table, you must specify the fields you want.
- 6.** Column names must be unique within a specific table. You may use the same column name in a different table.
- 7.** When a field is defined as “`NOT NULL`” this means that field value can never be empty. Each time a new row is added, you must enter a value in that field.

## Chapter Glossary

**Alias:** An abbreviated name for a database object to save keystrokes while writing queries.

**BCP:** Bulk Copy Program is a command-line utility for importing or exporting data to or from a delimited text file

**Clause:** A type of keyword used for a query. SELECT is both a clause and a keyword.

**Database:** A collection of objects to store and retrieve data.

**Database Context:** Refers to which database you are running the current query against.

**Data type:** An attribute of a field that tells SQL Server what kind of data that field may accept. Examples include integers, dates and characters.

**Delimiter:** A character that separates one object or entity from another.

**Field:** A column in a database table.

**Inner Join:** Combines records from two or more tables where matching values are found.

**Keyword:** A word built into the T-SQL language.

**Management Studio:** The user interface to talk with SQL Server services.

**Operator:** Words or symbols used by T-SQL to make calculations or evaluations. Some commonly used operators are “AND,” “OR,” (<), (>) and (=).

**Outer Join:** Combines records from two or more tables and shows matching and mismatching values.

**Populated:** A term used to describe a table that contains data.

**Primary Key:** Attribute of a field that ensures no two records have an identical value for that field.

**Query:** A question you ask to get information from data in database.

**Record:** A row of data in a table.

**Record set:** The set of data returned as an answer to a query.

**Result set:** Another term for record set.

**Service:** A process, much like an application, that runs in the background of your system.

**SQL:** Structured Query Language.

**T-SQL:** Transact Structured Query Language is the computer programming language based on the SQL standard and used by Microsoft SQL Server to create databases, populate tables and retrieve data.

## Chapter One - Review Quiz

1.) Which record will not return with the following WHERE clause?

**WHERE LastName like 'T%'**

- O a. Thomas
- O b. Atwater
- O c. Tompter.
- O d. TeeTee

2.) Look at the following T-SQL statement:

```
SELECT *  
FROM Employee LEFT OUTER JOIN Location  
ON Location.LocationID = Employee.LocationID
```

What will be displayed in the result set?

- O a. All records where both tables match.
- O b. All records in Employee including matches from Location.
- O c. All records from Location including matches from Employee.

3.) What is the correct way to alias the sales table?

- O a. FROM Sales AS sl
- O b. FROM sl AS Sales sl
- O c. SELECT Sales AS sl

4.) You have a table named Employee. You write the following query:

```
SELECT *  
FROM Employee em
```

You plan to join the Location table and fear there may be some employees with no location. You want to make sure that the query returns a list of all employee records. What join clause would you add to the query above?

- O a. LEFT JOIN Location lo ON em.LocationID = lo.LocationID
- O b. RIGHT JOIN Location lo ON em.LocationID = lo.LocationID
- O c. INNER JOIN Location lo ON em.LocationID = lo.LocationID
- O d. FULL JOIN Location lo ON em.LocationID = lo.LocationID

5.) BCP is a command-line utility that does what?

- O a. It runs SQL scripts.
- O b. It installs SQL Server.
- O c. It imports data from any type of file.
- O d. It imports data from a text file.

6.) If you do not specify a way to delimit your fields what does BCP do?

- O a. You get an error message.
- O b. It picks the most recently used delimiting option.
- O c. You don't get any delimited file.
- O d. You get a comma delimited file.
- O e. You get a tab delimited file.

7.) You want to find all first names that start with the letters A-M in your Customer table. Which T-SQL code would you use?

- O a. `SELECT * FROM Customer  
WHERE Firstname <= 'm%'`
- O b. `SELECT * FROM Customer  
WHERE Firstname = 'a-m%'`
- O c. `SELECT * FROM Customer  
WHERE Firstname LIKE 'a-m%'`
- O d. `SELECT * FROM Customer  
WHERE Firstname = '[a-m]%'`
- O e. `SELECT * FROM Customer  
WHERE Firstname LIKE '[a-m]%'`

8.) You want to find all scores for contestants who scored in the range of 20-30 points. Which T-SQL code would you use?

- O a. `SELECT * FROM contestant  
WHERE score BETWEEN 20 OR 30`
- O b. `SELECT * FROM contestant  
WHERE score BETWEEN 20 AND 30`
- O c. `SELECT * FROM contestant  
WHERE score IS BETWEEN 20 AND 30`
- O d. `SELECT * FROM contestant  
WHERE score MIDDLE RANGE (20,30)`

9.) You want to find all first names that have the letter A as the second letter and do not end with the letter Y. Which T-SQL code would you use?

- O a.     SELECT \* FROM Employee  
          WHERE Firstname LIKE '\_A%' AND Firstname NOT LIKE 'Y%'
- O b.     SELECT \* FROM Employee  
          WHERE Firstname LIKE '\_A%' AND Firstname NOT LIKE '%Y'
- O c.     SELECT \* FROM Employee  
          WHERE Firstname LIKE 'A\_%' AND Firstname NOT LIKE 'Y%'

10.) You work for a commercial CA (certificate authority) which helps identify trusted third party URLs. You have a table named ApprovedWebSites. Some are ftp:// sites and some are http:// sites. You want to find all approved .org sites listed in the URLName field of the ApprovedWebSites table. All URL names will have the :// with at least one character before them. All sites will have at least one character after the :// and at least one character before the .org at the end. What code will give you all .org records?

- O a.     SELECT \* FROM ApprovedWebSites  
          WHERE URLName LIKE '%://\_[.org]'
- O b.     SELECT \* FROM ApprovedWebSites  
          WHERE URLName LIKE '\_%[.org]'
- O c.     SELECT \* FROM ApprovedWebSites  
          WHERE URLName LIKE '\_%://%.org'
- O d.     SELECT \* FROM ApprovedWebSites  
          WHERE URLName = '%://%.org'

## Answer Key

1.) Because % represents zero or more characters and the first character in the pattern to be matched is 'T' the only names that will be returned will start with 'T'. Since 3 of the names do start with 'T' (a) Thomas, (c) Tompter and (d) TeeTee are all wrong. Since Atwater is the only one that does not start with 'T' (b) is the correct answer.

2.) An INNER JOIN will return 'All records where both tables match' so (a) is incorrect. RIGHT OUTER JOINing to Location would return 'All records from Location including matches from Employee making (c) wrong too. Because

Employee is on the left and Location is on the right of the LEFT OUTER JOIN operator (b) is correct and will return 'All records in Employee including matches from Location'

3.) 'FROM sl AS Sales sl' will alias the 'sl' tables as 'Sales' then display a syntax error because of the second 'sl' making (b) wrong. 'SELECT Sales AS sl' will alias the 'Sales' field as 'sl' so (c) is also wrong. 'FROM Sales AS sl' will alias the 'Sales' table as 'sl' so (a) is correct.

4.) RIGHT JOIN Location lo ON em.LocationID = lo.LocationID will return all location records including any matches from the employee table so (b) is wrong. INNER JOIN Location lo ON em.LocationID = lo.LocationID will only return the employee records that have a matching location so (c) is wrong too. FULL JOIN Location lo ON em.LocationID = lo.LocationID will return the superset of both tables making (d) incorrect. LEFT JOIN Location lo ON em.LocationID = lo.LocationID will return all employee records including any matches from the location table making (a) the correct answer.

5.) SQL Server runs SQL scripts so (a) is incorrect. Installation discs are used for installing SQL Server so (b) is wrong too. Programs like SQL Server Integration Services (SSIS) can import data from most types of files so (c) is also wrong. The correct answer is (d) because BCP (Bulk Copy Program) imports data from text files.

6.) Since, by default, you get a tab delimited file when you do not specify a way to delimit your fields using the -t switch then (a), (b), (c) and (d) are all wrong. BCP will create a tab delimited file if the -t switch is not used making (e) the correct answer.

7.) Wildcards (% , \_) only work with the LIKE keyword making (a), (b) and (d) all incorrect. 'LIKE 'a-m%'' would only match strings where the first three characters are 'a-m' so (c) is also wrong. The correct answer is (e) because the predicate uses the LIKE keyword and ends in %, meaning zero or more characters following the first character, which has its range defined correctly with [a-m].

8.) BETWEEN uses the AND operator so (a) is incorrect because it used the OR operator. The IS operator is not used with the BETWEEN operator so (c) is also incorrect. There is no such operator as MIDDLE RANGE so (d) is wrong too. The BETWEEN operator requires the AND operator but doesn't need the IS operator so (b) is correct.

9.) When pattern matching, the use of NOT LIKE 'Y%' only ensures that the first character in the string is not 'Y' it does not check to see what the string ends in so (a) and (c) are both wrong. The use of LIKE '\_A%' will match strings with any character in the first position followed by 'A' followed by zero or more

characters; the use of NOT LIKE '%Y' ensures that the last character in the string is not 'Y' so (b) is correct.

10.) The wildcard '%' can match zero characters so (a) and (d) are both wrong because they would match a string starting with ':/'. The pattern we need to match must include ':/' and (b) doesn't include this so it is wrong too. The two wildcards '\_' and '%' used side by side ensure that a pattern contains at least one character immediately followed by zero or more characters; therefore (c) is the correct answer.

## Bug Catcher Game

To play the Bug Catcher game run the SQLQueriesBugCatcherCh1.pps from the BugCatcher folder of the companion files. You can obtain these files from [www.Joes2Pros.com](http://www.Joes2Pros.com) or by ordering the Companion CD.

## Chapter 2. Query Options

Have you ever been given a vague answer to a question? Let's say you asked someone where they worked. You are looking for perhaps a city name or address and they tell you, "I work at headquarters." It's an accurate answer, but not the detailed answer you wanted to know. Even when you get all the correct information, it may be contained within a large report. Perhaps the results need to be sorted alphabetically by name, or in order by revenue amounts, so the highest or lowest data will stand out at the top of your list.

In this chapter, we will encounter similar issues trying to get table data to answer the questions we ask, as well as arranging data in the order we'd like to view the information. You will work with multi-table joins every day of your career working with SQL Server and data. As well, the sorting techniques introduced in this chapter are key skills you will need in Chapters 4 through 7.

**READER NOTE:** Please run the script *SQLQueriesChapter2.0Setup.sql* in order to follow along with the examples in the first section of Chapter 2. All scripts mentioned in this chapter may be found at [www.Joes2Pros.com](http://www.Joes2Pros.com).

## Sorting Data

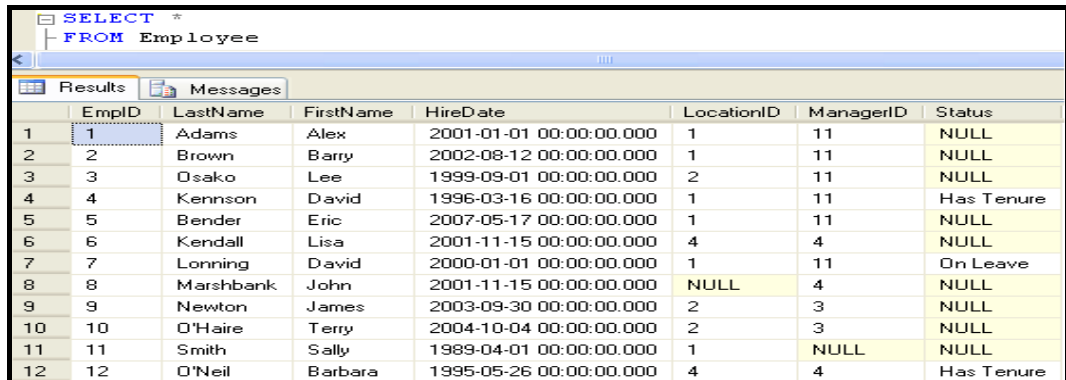
Now that you've mastered building basic queries with T-SQL, you are eager to power forward in your understanding SQL Server. Beware – at first glance the sorting topics in this section may seem like optional finishing touches. Up until now, the result sets we've built have been small enough to visualize all records in one glance and to answer questions without needing to rearrange the order of the data. However, data sorting is one capability that differentiates SQL Joes from SQL Pros. Two things you will encounter in a data intense job setting are fast motivators to learn sorting: 1) very large record sets (hundreds, thousands, even millions of records!), and 2) managers and decisionmakers who are very specific and require precise appearance of information in your reports. *In short, the ability to properly sort data can keep you out of hot water!*

Just like table joins and filtering, when you sort data it affects only the view of the information in your result set. *Sorting does not physically change the way the underlying table stores the data.* Sorting is a key step in turning your data into report information.

You can sort any field in ascending or descending order. Ascending (keyword ASC) will order characters alphabetically A-Z; numbers from lowest to highest; and dates from oldest to newest. Descending (keyword DESC) does just the opposite and gives you the newest dates, highest numbers, and a reverse alphabetical sort Z-A. SQL Server's default sort order is ascending, so your result will appear in ascending order if you don't specify.

## Sorting Single Table Queries

Let's begin by simply selecting all the records in the Employee table of the JProCo database without specifying a sort order (see Figure 2.1).



	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	1	Adams	Alex	2001-01-01 00:00:00.000	1	11	NULL
2	2	Brown	Barry	2002-08-12 00:00:00.000	1	11	NULL
3	3	Osako	Lee	1999-09-01 00:00:00.000	2	11	NULL
4	4	Kennson	David	1996-03-16 00:00:00.000	1	11	Has Tenure
5	5	Bender	Eric	2007-05-17 00:00:00.000	1	11	NULL
6	6	Kendall	Lisa	2001-11-15 00:00:00.000	4	4	NULL
7	7	Lonning	David	2000-01-01 00:00:00.000	1	11	On Leave
8	8	Marshbank	John	2001-11-15 00:00:00.000	NULL	4	NULL
9	9	Newton	James	2003-09-30 00:00:00.000	2	3	NULL
10	10	O'Haire	Terry	2004-10-04 00:00:00.000	2	3	NULL
11	11	Smith	Sally	1989-04-01 00:00:00.000	1	NULL	NULL
12	12	O'Neil	Barbara	1995-05-26 00:00:00.000	4	4	Has Tenure

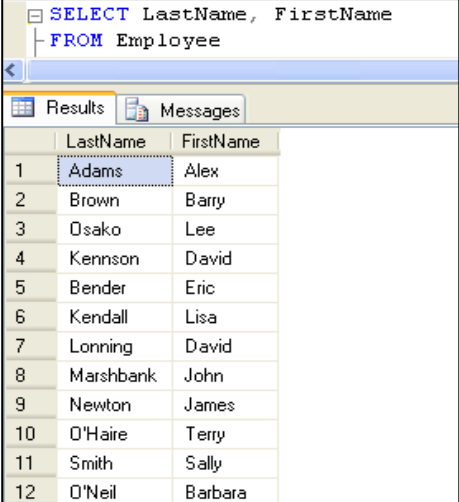
Figure 2.1 SQL Server naturally sorts by an indexed field, such as EmpID.

## Chapter 2. Query Options

Note SQL Server naturally sorts this table on the EmpID field (see Figure 2.1). If there is no ID field or indexed field, table data will generally display in the order the records were created. Now change your Select statement to include just LastName and FirstName (see Figure 2.2). Notice that SQL Server still orders the records by this table's natural sort (EmpID).

To sort on the LastName field (ascending), add an ORDER BY clause after the FROM clause in your previous query (see Figure 2.3).

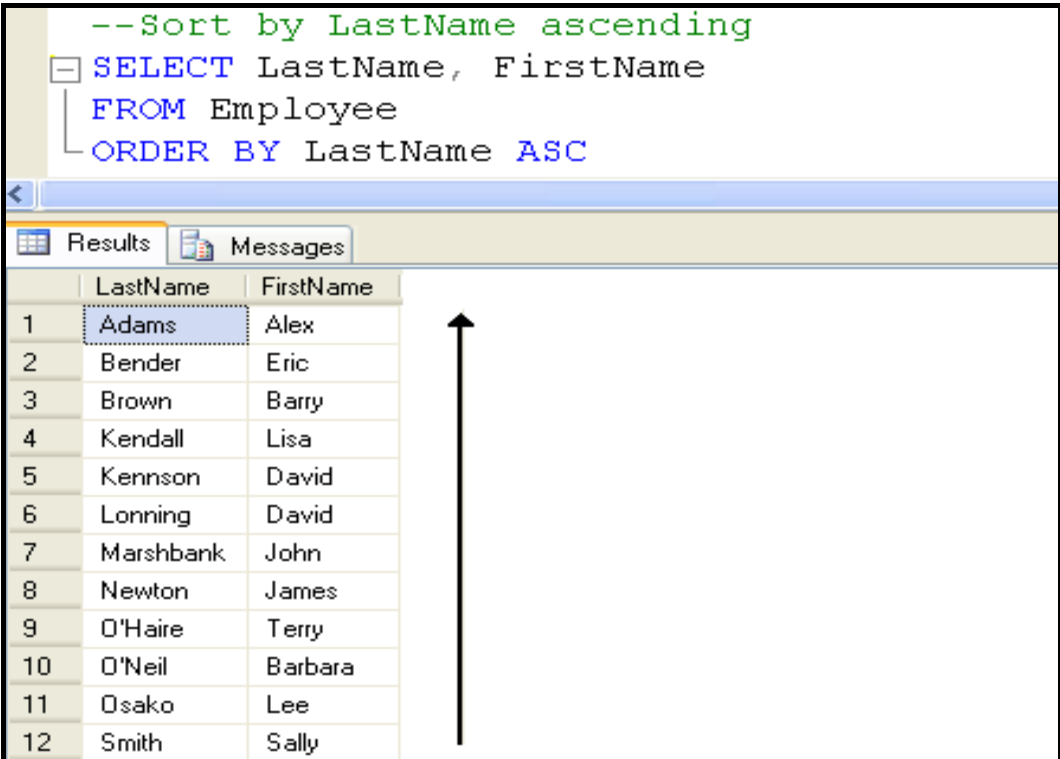
When specifying sort order SQL Server will not recognize the full words "ascending" or "descending." Therefore, you must use the keywords ASC and DESC.



```
SELECT LastName, FirstName
FROM Employee
```

	LastName	FirstName
1	Adams	Alex
2	Brown	Barry
3	Osako	Lee
4	Kennson	David
5	Bender	Eric
6	Kendall	Lisa
7	Lonning	David
8	Marshbank	John
9	Newton	James
10	O'Haire	Terry
11	Smith	Sally
12	O'Neil	Barbara

**Figure 2.2** SQL Server still orders by EmpID, even if the result set does not show EmpID.



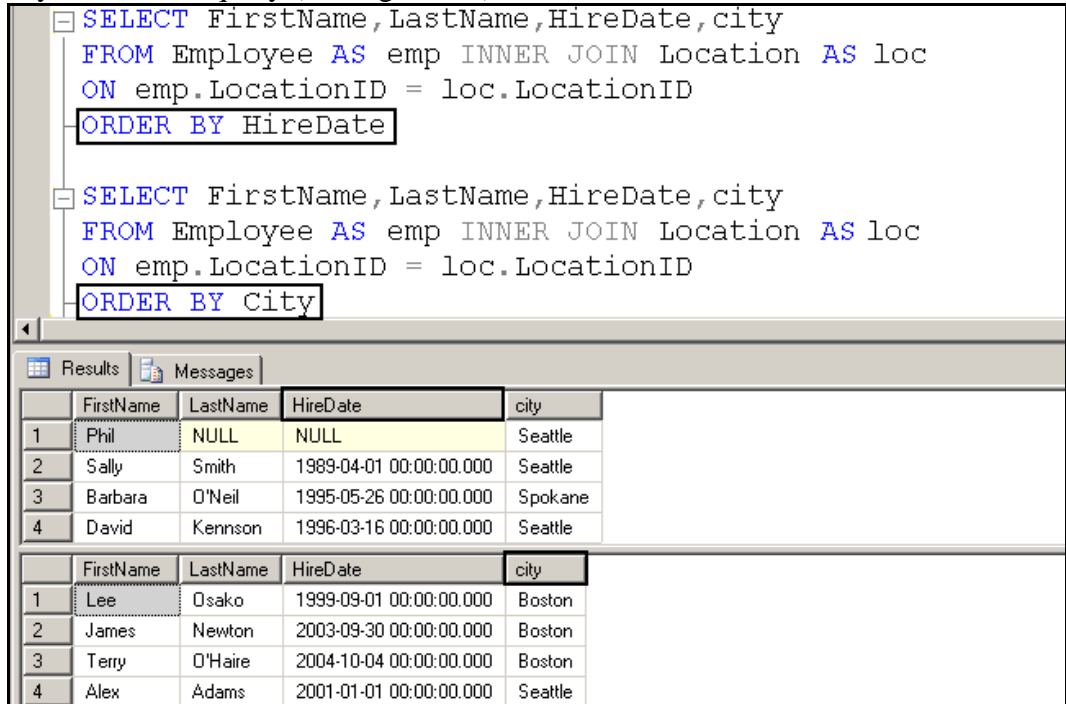
```
--Sort by LastName ascending
SELECT LastName, FirstName
FROM Employee
ORDER BY LastName ASC
```

	LastName	FirstName
1	Adams	Alex
2	Bender	Eric
3	Brown	Barry
4	Kendall	Lisa
5	Kennson	David
6	Lonning	David
7	Marshbank	John
8	Newton	James
9	O'Haire	Terry
10	O'Neil	Barbara
11	Osako	Lee
12	Smith	Sally

**Figure 2.3** The Employee data now sorted by LastName (ascending). Compare to Figure 2.2.

## Sorting Multiple Table Queries

When you join two or more tables in a query, you can then sort on any field of any table in the query (see Figure 2.4).



```

SELECT FirstName, LastName, HireDate, city
FROM Employee AS emp INNER JOIN Location AS loc
ON emp.LocationID = loc.LocationID
ORDER BY HireDate

SELECT FirstName, LastName, HireDate, city
FROM Employee AS emp INNER JOIN Location AS loc
ON emp.LocationID = loc.LocationID
ORDER BY City

```

	FirstName	LastName	HireDate	city
1	Phil	NULL	NULL	Seattle
2	Sally	Smith	1989-04-01 00:00:00.000	Seattle
3	Barbara	O'Neil	1995-05-26 00:00:00.000	Spokane
4	David	Kenison	1996-03-16 00:00:00.000	Seattle

	FirstName	LastName	HireDate	city
1	Lee	Osako	1999-09-01 00:00:00.000	Boston
2	James	Newton	2003-09-30 00:00:00.000	Boston
3	Terry	O'Haire	2004-10-04 00:00:00.000	Boston
4	Alex	Adams	2001-01-01 00:00:00.000	Seattle

Figure 2.4 When joining multiple tables, you can sort on any field.

## Sorting Data With Nulls

Oftentimes the fields you sort will contain null values. *Null always sorts before any data.* So in an ascending query, the nulls always show first. In a descending query, the nulls always show last.

Open a new query window to the JProCo database, paste in the following code, and observe the different way each query handles null values (see Figure 2.5).

**Status in DESC order, nulls appear last**

```

SELECT *
FROM Employee
ORDER BY Status DESC

```

**Status defaults to ASC, nulls appear first**

```

SELECT *
FROM Employee
ORDER BY Status

```

	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	7	Lonning	David	2000-01-01	1	11	On Leave
2	4	Kennson	David	1996-03-16	1	11	Has Tenure
3	12	O'Neil	Barbara	1995-05-26	4	4	Has Tenure
4	5	Bender	Eric	2007-05-17	1	11	NULL
5	6	Kendall	Lisa	2001-11-15	4	4	NULL
6	1	Adams	Alex	2001-01-01	1	11	NULL
7	2	Brown	Barry	2002-08-12	1	11	NULL
8	3	Osako	Lee	1999-09-01	2	11	NULL
9	8	Marshbank	John	2001-11-15	NULL	4	NULL
10	9	Newton	James	2003-09-30	2	3	NULL
11	10	O'Haire	Terry	2004-10-04	2	3	NULL
12	11	Smith	Sally	1989-04-01	1	NULL	NULL

	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	1	Adams	Alex	2001-01-01	1	11	NULL
2	2	Brown	Barry	2002-08-12	1	11	NULL
3	3	Osako	Lee	1999-09-01	2	11	NULL
4	8	Marshbank	John	2001-11-15	NULL	4	NULL
5	9	Newton	James	2003-09-30	2	3	NULL
6	10	O'Haire	Terry	2004-10-04	2	3	NULL
7	11	Smith	Sally	1989-04-01	1	NULL	NULL
8	5	Bender	Eric	2007-05-17	1	11	NULL
9	6	Kendall	Lisa	2001-11-15	4	4	NULL
10	12	O'Neil	Barbara	1995-05-26	4	4	Has Tenure
11	4	Kennson	David	1996-03-16	1	11	Has Tenure
12	7	Lonning	David	2000-01-01	1	11	On Leave

**Figure 2.5** The effect of DESC vs. ASC (default) sorting on a field containing null values.

Running both queries at once gives you the two result sets you see in Figure 2.5. The first query is descending and puts the nulls last, and the second query shows the nulls first.

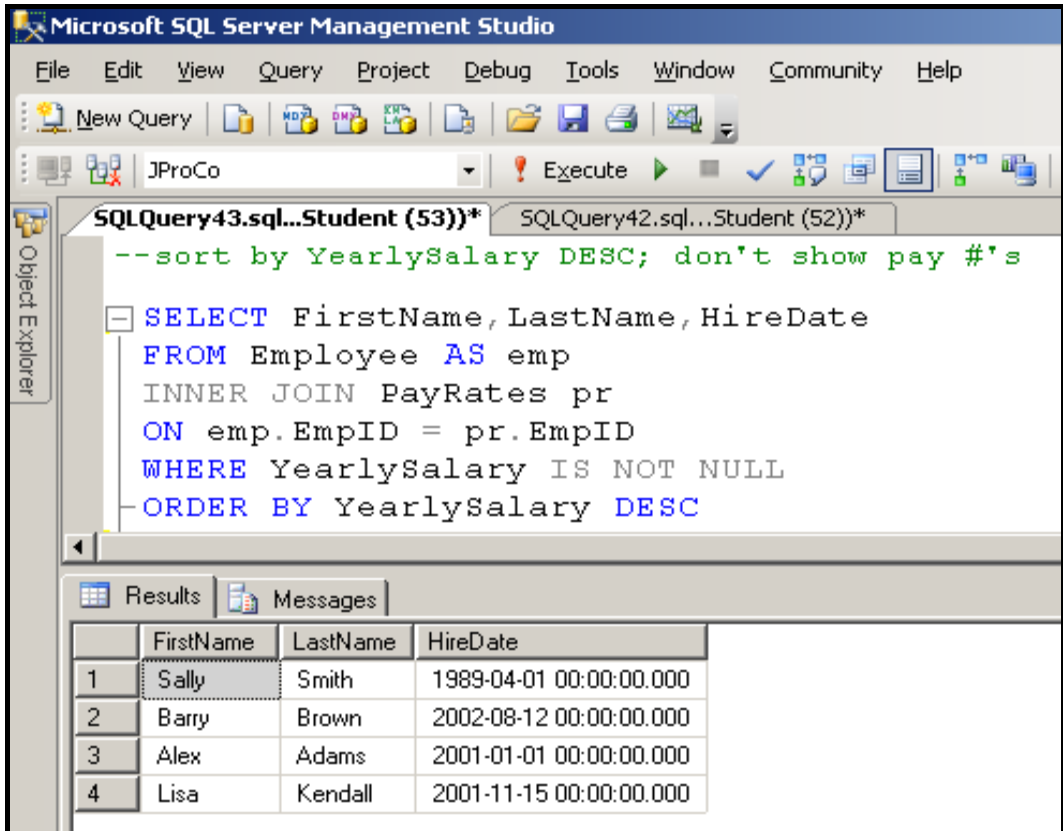
## Sorting Hidden Results

In your SQL Server career, you will frequently encounter reports where you must sort by a field not displayed in your final report. For example, you might be asked to show an employee list sorted by highest to lowest pay. However, because of confidentiality reasons, you can't show any actual pay numbers in the report.

You can sort your results on any field in your query, including field(s) not contained in the SELECT list. This example sorts by the YearlySalary field but doesn't show YearlySalary in the result set:

## Chapter 2. Query Options

We have four employees who receive a yearly salary. Of those employees, Sally Smith is the highest paid employee and Lisa Kendall is the lowest. In Figure 2.6 we see these employees in order while not revealing any of the salary fields in our report.



The screenshot shows the Microsoft SQL Server Management Studio interface. The query editor displays the following SQL code:

```
--sort by YearlySalary DESC; don't show pay #'s  
  
SELECT FirstName, LastName, HireDate  
FROM Employee AS emp  
INNER JOIN PayRates pr  
ON emp.EmpID = pr.EmpID  
WHERE YearlySalary IS NOT NULL  
ORDER BY YearlySalary DESC
```

The Results pane shows the following data:

	FirstName	LastName	HireDate
1	Sally	Smith	1989-04-01 00:00:00.000
2	Barry	Brown	2002-08-12 00:00:00.000
3	Alex	Adams	2001-01-01 00:00:00.000
4	Lisa	Kendall	2001-11-15 00:00:00.000

**Figure 2.6** We can sort by YearlySalary without displaying the field in our result set.

## Sorting Levels

The ORDER BY clause allows you to sort using multiple fields. In Figure 2.7 the sort is done first on City, so all three Boston employees appear first in the result set. Within the Boston group of records Newton comes before O’Haire and Osako is the last Boston Employee. The next group is Seattle, which has six records in its group. Adams is first and Smith is last.

If you choose a sort field which includes the same value for multiple records, then SQL Server will display those “tie” records in their natural order. If you apply two fields in a sort, then your second field acts as a “tie breaker.”

In the Grant table, two grants each show an amount of \$21,000. Figure 2.8 illustrates the difference a secondary sort field (GrantName) makes when sorting the Grant table by Amount.

```

SELECT FirstName, LastName,
city, [State]
FROM Employee AS emp
INNER JOIN Location AS loc
ON emp.LocationID = loc.LocationID
ORDER BY City, LastName
    
```

FirstName	LastName	city	State
James	Newton	Boston	MA
Terry	O'Haire	Boston	MA
Lee	Osako	Boston	MA
Alex	Adams	Seattle	WA
Eric	Bender	Seattle	WA
Barry	Brown	Seattle	WA
David	Kennson	Seattle	WA
David	Lonning	Seattle	WA
Sally	Smith	Seattle	WA
Lisa	Kendall	Spokane	WA
Barbara	O'Neil	Spokane	WA

Figure 2.7 You can sort on multiple fields.

```

SELECT *
FROM [Grant]
ORDER BY Amount DESC, GrantName
    
```

GrantID	GrantName	EmpID	Amount
007	Ben@MoreTechnology.com	10	41000.00
008	www.@-Last-U-Can-Help.com	7	25000.00
009	Thank you @.com	11	21500.00
005	BIG G's Foundation%	4	21000.00
004	Norman's Outreach	NULL	21000.00
003	Robert@BigStarBank.com	7	18100.00
006	TALTA_Kishan International	3	18100.00
002	K_Land fund trust	2	15750.00
010	Call Mom @Com	5	7500.00
001	92 Purr_Scents %% team	7	4750.00

Figure 2.8 The secondary sort on this table is by GrantName.

## Lab 2.1: Sorting Data

**Lab Prep:** Before you can begin the lab you must run the SQLQueriesChapter2.1Setup.sql script. It is recommended that you view the lab video instructions found in Lab2.1\_SortingData.wmv, available at [www.Joe2Pros.com](http://www.Joe2Pros.com).

**Skill Check 1:** Show all records from the Grant table sorted alphabetically by GrantName. Your result should look like Figure 2.9 (below).

	GrantID	GrantName	EmplID	Amount
1	001	92 Purr_Scents %% team	7	4750.00
2	007	Ben@MoreTechnology.com	10	41000.00
3	005	BIG 6's Foundation%	4	21000.00
4	010	Call Mom @Com	5	7500.00
5	002	K_Land fund trust	2	15750.00
6	004	Norman's Outreach	NULL	21000.00
7	003	Robert@BigStarBank.com	7	18100.00
8	006	TALTA_Kishan International	3	18100.00
9	009	Thank you @.com	11	21500.00
10	008	www.@-Last-U-Can-Help....	7	25000.00

(local) (10.0 RTM) | RENO\Student (53) | JProCo | 00:00:00 | 10 rows

Ln 3      Col 18      Ch 18      INS

**Figure 2.9** The result of Skill Check 1 shows 10 records sorted by GrantName.

**Skill Check 2:** Show all fields from the Employee table. The most recent HireDate should appear first. (Figure 2.10)

	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	5	Bender	Eric	2007-05-17 ...	1	11	NULL
2	10	O'Haire	Terry	2004-10-04 ...	2	3	NULL
3	9	Newton	James	2003-09-30 ...	2	3	NULL
4	2	Brown	Barry	2002-08-12 ...	1	11	NULL
5	6	Kendall	Lisa	2001-11-15 ...	4	4	NULL
6	8	Marshbank	John	2001-11-15 ...	NULL	4	NULL
7	1	Adams	Alex	2001-01-01 ...	1	11	NULL
8	7	Lonning	David	2000-01-01 ...	1	11	On Leave
9	3	Osako	Lee	1999-09-01 ...	2	11	NULL
10	4	Kennson	David	1996-03-16 ...	1	11	Has Tenure
11	12	O'Neil	Barbara	1995-05-26 ...	4	4	Has Tenure
12	11	Smith	Sally	1989-04-01 ...	1	NULL	NULL

Query executed successfully (local) (10.0 RTM) RENO\Student (53) JProCo 00:00:00 12 rows

**Figure 2.10** Skill Check 2 shows the most recently hired person listed first

**Skill Check 3:** Query the CurrentProducts table for just the ProductName and Category fields. Sort the table by the most expensive RetailPrice on top and the least on the bottom. When you're done, your result should resemble the figure you see here (Figure 2.11 shows the first 5 of 480 rows).

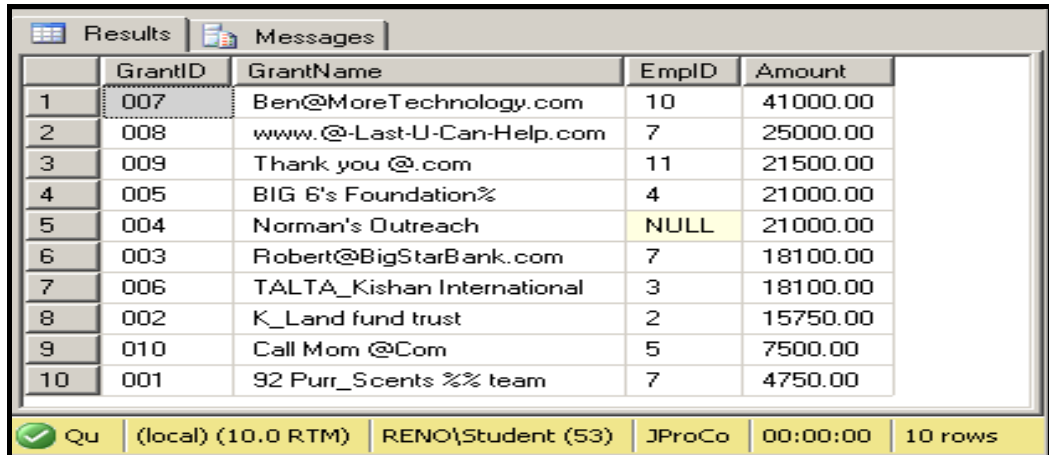
	ProductName	Category
1	Lakes Tour 2 Weeks West Coast	LongTerm-Stay
2	Lakes Tour 2 Weeks East Coast	LongTerm-Stay
3	Rain Forest Tour 2 Weeks East Coast	LongTerm-Stay
4	River Rapids Tour 2 Weeks East Coast	LongTerm-Stay
5	Wine Tasting Tour 2 Weeks West Coast	LongTerm-Stay

(local) (10.0 RTM) RENO\Student (53) JProCo 00:00:00 480 rows

**Figure 2.11** Show the most expensive products first without showing RetailPrice in the Select list.

## Chapter 2. Query Options

**Skill Check 4:** Now sort all the fields of the [Grant] table from highest to lowest amount. If any Grants have a tying amount, then list the ties alphabetically by GrantName (see Figure 2.12).



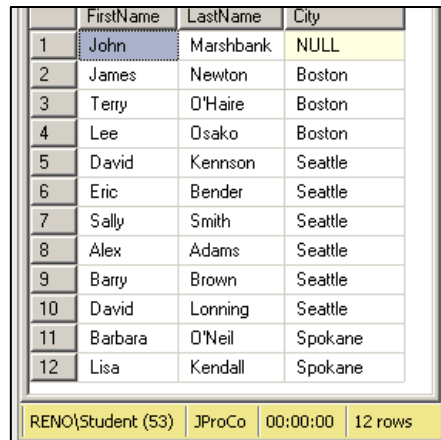
	GrantID	GrantName	EmplD	Amount
1	007	Ben@MoreTechnology.com	10	41000.00
2	008	www.@-Last-U-Can-Help.com	7	25000.00
3	009	Thank you @.com	11	21500.00
4	005	BIG 6's Foundation%	4	21000.00
5	004	Norman's Outreach	NULL	21000.00
6	003	Robert@BigStarBank.com	7	18100.00
7	006	TALTA_Kishan International	3	18100.00
8	002	K_Land fund trust	2	15750.00
9	010	Call Mom @Com	5	7500.00
10	001	92 Purr_Scents %% team	7	4750.00

Results Messages

Qu (local) (10.0 RTM) RENO\Student (53) JProCo 00:00:00 10 rows

**Figure 2.12** Skill Check 4 shows the highest amounts listed first. Where two grants have the same amount (\$21,000), Big 6 is listed before Norman's because the secondary sort is alphabetical by GrantName.

**Skill Check 5:** Join the Employee and Location tables together in an OUTER JOIN that shows all the employee records even if they have no location. Show the fields FirstName, LastName, and City. Sort your result so that null city names appear first and the remaining values appear in ascending order (see Figure 2.13)



	FirstName	LastName	City
1	John	Marshbank	NULL
2	James	Newton	Boston
3	Terry	O'Haire	Boston
4	Lee	Osako	Boston
5	David	Kennson	Seattle
6	Eric	Bender	Seattle
7	Sally	Smith	Seattle
8	Alex	Adams	Seattle
9	Barry	Brown	Seattle
10	David	Lonning	Seattle
11	Barbara	O'Neil	Spokane
12	Lisa	Kendall	Spokane

RENO\Student (53) JProCo 00:00:00 12 rows

**Figure 2.13** Skill Check 5 shows Employee names and city listed in order by city with Nulls appearing first.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab2.1\_SortingData.sql

## Sorting Data - Points to Ponder

- 1.** The ORDER BY clause enables you to sort your query results.
- 2.** You can append the DESC (descending) and ASC (ascending) keywords to your ORDER BY clause.
- 3.** If you do not specify DESC or ASC, then the default is to sort ascending.
- 4.** SQL Server will not recognize the full words “descending” or “ascending” – you must specify the keywords as DESC or ASC.
- 5.** You can sort by more than one field. This is useful when the primary sort field has many identical values.
- 6.** If null values appear in your sort, they are first in ASC queries and last in DESC queries.
- 7.** SQL Server’s natural sort order is first by indexed fields (the primary key field or an ID field). If no such fields are present, then SQL Server displays records in the order they were created.
- 8.** In a simple query you choose the field(s) you wish to sort on.
- 9.** The ORDER BY clause generally appears last.
- 10.** You can sort on any field in your query even if you leave that field out of the select list.

## Exploring Related Tables

A relational database contains many related tables. These two tables relate on the LocationID field (see Figure 2.14).

The screenshot shows a SQL query window with the following SQL code:

```
SELECT * FROM Employee
SELECT * FROM Location
```

The results are displayed in two tables. The top table is the Employee table, and the bottom table is the Location table. The LocationID field is highlighted in yellow in both tables to show the relationship.

	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	1	Adams	Alex	2001-01-01 00:00:00.000	1	11	NULL
2	2	Brown	Barry	2002-08-12 00:00:00.000	1	11	NULL
3	3	Osako	Lee	1999-09-01 00:00:00.000	2	11	NULL
4	4	Kennson	David	1996-03-16 00:00:00.000	1	11	Has Tenure
5	5	Bender	Eric	2007-05-17 00:00:00.000	1	11	NULL
6	6	Kendall	Lisa	2001-11-15 00:00:00.000	4	4	NULL
7	7	Lonning	David	2000-01-01 00:00:00.000	1	11	On Leave
8	8	Marshbank	John	2001-11-15 00:00:00.000	NULL	4	NULL
9	9	Newton	James	2003-09-30 00:00:00.000	2	3	NULL
10	10	O'Haire	Terry	2004-10-04 00:00:00.000	2	3	NULL
11	11	Smith	Sally	1989-04-01 00:00:00.000	1	NULL	NULL
12	12	O'Neil	Barbara	1995-05-26 00:00:00.000	4	4	Has Tenure

	LocationID	street	city	state
1	1	111 First ST	Seattle	WA
2	2	222 Second AVE	Boston	MA
3	3	333 Third PL	Chicago	IL
4	4	444 Ruby ST	Spokane	WA

**Figure 2.14** These two tables relate on the LocationID field.

This gives us the ability to take these same tables and put them into the same query using a join. We also have the ability to join up to 256 tables in one query.

We can see that Alex Adams works in Location 1, which is in Seattle. Barry Brown also works in Location 1. Lee Osako works in Location 2, which is on 2nd Avenue in Boston. So this second table referencing Location is known as a **lookup table**. It contains more detailed information about the LocationID as referenced from the Employee table.

Let's next look at the Grant table, where we can see Ben@MoreTechnology.com was the largest grant at \$41,000 (Figure 2.15).

The screenshot shows a SQL query window with the following query: `SELECT * FROM [Grant]`. Below the query, there are tabs for 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with 10 rows and 4 columns: GrantID, GrantName, EmpID, and Amount. The first row is highlighted.

	GrantID	GrantName	EmpID	Amount
1	001	92 Purr_Scents % team	7	4750.00
2	002	K_Land fund trust	2	15750.00
3	003	Robert@BigStarBank.com	7	18100.00
4	004	Norman's Outreach	NULL	21000.00
5	005	BIG 6's Foundation%	4	21000.00
6	006	TALTA_Kishan International	3	18100.00
7	007	Ben@MoreTechnology.com	10	41000.00
8	008	www.@-Last-U-Can-Help.com	7	25000.00
9	009	Thank you @.com	11	21500.00
10	010	Call Mom @Com	5	7500.00

Figure 2.15 All records from the Grant table.

The employee who procured the grant from Ben@MoreTechnology.com was EmpID 10. What if we wanted to find more information about this employee? Is this employee a man or woman? When was this employee hired? In which of JProCo's four locations does this employee work?

To answer detailed questions about this employee, we need to look beyond the Grant table. Expand JProCo in the Object Explorer and look at the other tables to find data relating to the Grant table fields (GrantID, GrantName, EmpID, Amount).

Employee is the only other table in the JProCo database to share a common field with the Grant table. The field Grant.EmpID matches with Employee.EmpID. In Chapter 1, we learned that two tables may be joined if they share a common field. In this case, EmpID links Grant and Employee.

**Note:** Having the same field name (EmpID) helped us quickly find the link and related table in this case. But even if the field names weren't exactly the same, this field could still link the two tables because the data and data type within EmpID is the same in both tables. In your database career, you frequently will find overlapping fields (containing the same data with the same data type) between tables, but the fields will have different names. For example, if the field in Grant had been called Grant.EmployeeID, it still would be a match with Employee.EmpID. Grant.EmployeeNumber would also still be a match with Employee.EmpID. As long as the data and data type in both fields match, you will be able to join the tables. We will explore this topic further in Chapter 8.

## Chapter 2. Query Options

Let's resume our quest for more insight to the individual who procured JProCo's largest grant. We've identified Employee as the other table that relates to Grant. Looking at EmpID 10 on both tables, we see Terry O'Haire is the employee who found the \$41,000 grant (Figure 2.16).

Employee Table					Grant Table				
	EmpID	lastname	firstname	hiredate		GrantID	GrantName	EmpID	Amount
1	1	Adams	Alex	2001-01-01 00:00:00	1	001	92 Purr_Scents %% team	7	4750.00
2	2	Brown	Barry	2002-08-12 00:00:00	2	002	K_Land fund trust	2	15750.00
3	3	Osako	Lee	1999-09-01 00:00:00	3	003	Robert@BigStarBank.com	7	18100.00
4	4	Kennson	David	1996-03-16 00:00:00	4	004	Norman's Outreach	NULL	21000.00
5	5	Bender	Eric	2007-05-17 00:00:00	5	005	BIG 6's Foundation%	4	21000.00
6	6	Kendall	Lisa	2001-11-15 00:00:00	6	006	TALTA_Kishan International	3	18100.00
7	7	Lonning	David	2000-01-01 00:00:00	7	007	Ben@MoreTechnology.com	10	41000.00
8	8	Marshbank	John	2001-11-15 00:00:00	8	008	@Last-U-Can-Help	7	25000.00
9	9	Newton	James	2003-09-30 00:00:00	9	009	Thank you @.com	11	21500.00
10	10	O'Haire	Terry	2004-10-04 00:00:00	10	010	Call Mom @Com	5	7500.00
11	11	Smith	Sally	1989-04-01 00:00:00	1	NULL	NULL		
12	12	O'Neil	Barbara	1995-05-26 00:00:00	4	4	Has Tenure		

**Figure 2.16** EmpID is the link between the Grant and Employee tables.

In the next section we will see these two tables joined by query (see Figure 2.17).

What else can the Employee table tell us about Terry O'Haire? Earlier we wondered when this employee had been hired, whether this is a man or woman, and in which location the employee works.

Besides EmpID, LastName, and FirstName, the Employee table contains these fields: HireDate, LocationID, ManagerID and Status. Our HireDate question is answered (10/4/2004). However, we can't tell from this table whether Terry is male or female. And while the Employee table tells us Terry works in Location 2, it doesn't explicitly tell us the city.

To find the name of the city where Terry O'Haire works, we must repeat the process of checking other tables in the database to find a link to location data. As was the case with the Grant and Employee tables, we luck out and quickly find the field LocationID is the logical link to the Location table. By looking up Location 2 in the Location table, we now know that Terry O'Haire works in Boston.

Our curiosity about the employee who secured JProCo's largest grant took us on a quick tour of JProCo and told us more about Terry O'Haire. But more importantly, in the process we logically linked three tables (Grant, Employee, Location) in order to answer our questions.

## Joining Two Tables

We know the Grant and Employee tables overlap on the EmpID field. (Said another way, these tables are *related* on EmpID.) We will now write an actual join of the Grant and Employee tables, which we logically joined when we wanted more information about the person who secured the largest grant (see Fig 2.16).

Begin by putting both tables in the FROM clause

```
SELECT *
FROM [Grant] Employee
```

Separate them with an INNER JOIN

```
SELECT *
FROM [Grant] INNER JOIN Employee
```

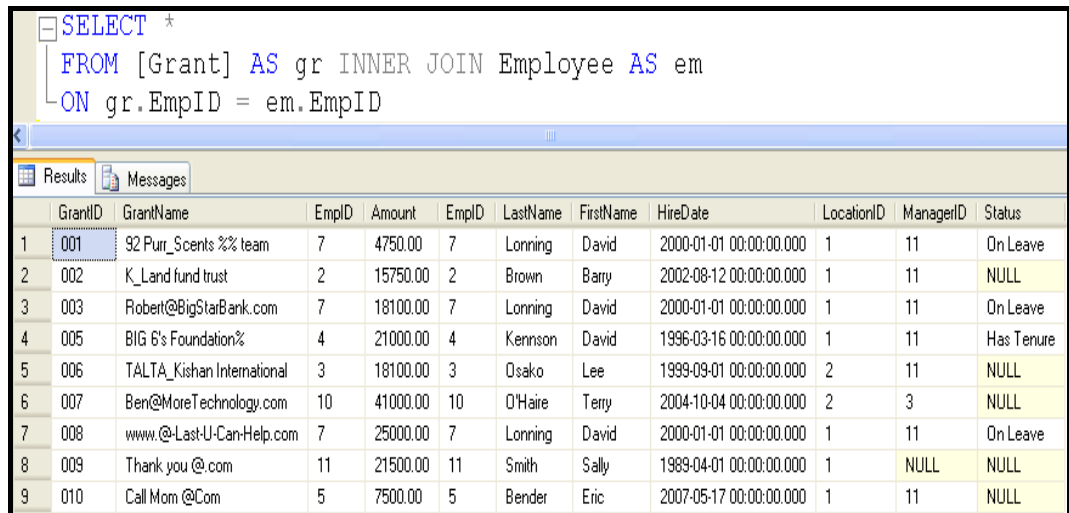
Alias the tables

```
SELECT *
FROM [Grant] AS gr INNER JOIN Employee AS em
```

Now define the field that joins them

```
SELECT *
FROM [Grant] AS gr INNER JOIN Employee AS em
ON gr.EmpID = em.EmpID
```

Now run the code. In each row, we see all fields from the Grant table and all fields for the employee who procured the grant (see Figure 2.17).



```
SELECT *
FROM [Grant] AS gr INNER JOIN Employee AS em
ON gr.EmpID = em.EmpID
```

	GrantID	GrantName	EmpID	Amount	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status
1	001	92.Purr_Scents%% team	7	4750.00	7	Lonning	David	2000-01-01 00:00:00.000	1	11	On Leave
2	002	K_Land fund trust	2	15750.00	2	Brown	Barry	2002-08-12 00:00:00.000	1	11	NULL
3	003	Robert@BigStarBank.com	7	18100.00	7	Lonning	David	2000-01-01 00:00:00.000	1	11	On Leave
4	005	BIG G's Foundation%	4	21000.00	4	Kennson	David	1996-03-16 00:00:00.000	1	11	Has Tenure
5	006	TALTA_Kishan International	3	18100.00	3	Osako	Lee	1999-09-01 00:00:00.000	2	11	NULL
6	007	Ben@MoreTechnology.com	10	41000.00	10	O'Haire	Terry	2004-10-04 00:00:00.000	2	3	NULL
7	008	www.@-Last-U-Can-Help.com	7	25000.00	7	Lonning	David	2000-01-01 00:00:00.000	1	11	On Leave
8	009	Thank you @.com	11	21500.00	11	Smith	Sally	1989-04-01 00:00:00.000	1	NULL	NULL
9	010	Call Mom @Com	5	7500.00	5	Bender	Eric	2007-05-17 00:00:00.000	1	11	NULL

Figure 2.17 An INNER JOIN on EmpID joins the Grant and Employee tables.

For information on the other types of joins, review the Chapter 1 coverage of Outer Joins, Left Outer Joins, Right Outer Joins, Full Outer Joins, and Mismatch Queries. Chapter 1 also contains a section on Table Aliasing.

## Joining Three Tables

So far we have seen examples querying a single table or two tables. In business and reporting settings, however, it's more common to see three or more tables joined together. (SQL Server can join up to 256 tables in a single query!)

Once you are proficient at joining two tables together, adding one more table to your query is not difficult. As we did with basic joins, we always begin building our query with a "SELECT \*" statement. Only after all needed tables are joined and we see them working properly do we narrow down our field selection list.

In the section "Exploring Related Tables," we logically joined three tables. Let's now do an actual join of those tables and see grant, employee, and location data combined within a single result set.

We've already joined the Grant and Employee tables. Aside from the EmpID field appearing twice, the result looks like a single table. Any of the fields you see here in Figure 2.18 may be used to join additional tables to this result.

Since LocationID appears in the combined Grant-Employee result, we know the Location table may be joined using the overlapping field (LocationID).

```

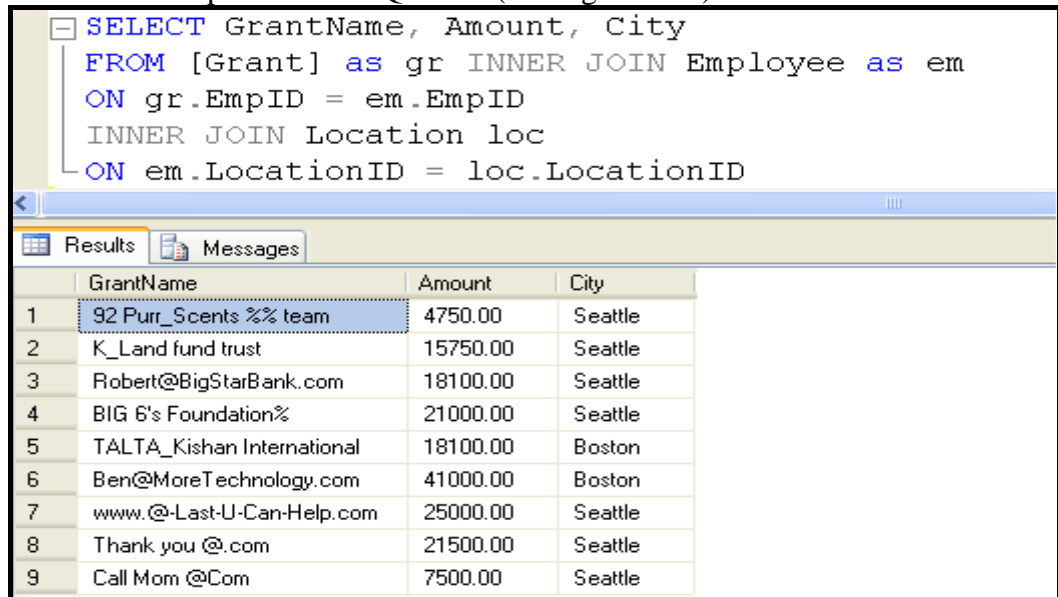
SELECT *
FROM [Grant] AS gr INNER JOIN Employee AS em
ON gr.EmpID = em.EmpID
INNER JOIN Location AS lo
ON lo.LocationID = em.LocationID
    
```

GrantID	GrantName	EmpID	Amount	EmpID	LastName	FirstName	HireDate	LocationID	ManagerID	Status	Location...	street	city	state	
1	001	92 Purr_Scents %%	7	4750.00	7	Lonning	David	2000-01-...	1	11	On Lea...	1	111 First ST	Seattle	WA
2	002	K_Land fund trust	2	15750.00	2	Brown	Barry	2002-08-...	1	11	NULL	1	111 First ST	Seattle	WA
3	003	Robert@BigStarBa...	7	18100.00	7	Lonning	David	2000-01-...	1	11	On Lea...	1	111 First ST	Seattle	WA
4	005	BIG 6's Foundation%	4	21000.00	4	Kenyson	David	1996-03-...	1	11	Has Te...	1	111 First ST	Seattle	WA
5	006	TALTA_Kishan Int...	3	18100.00	3	Osako	Lee	1999-09-...	2	11	NULL	2	222 Seco...	Boston	MA
6	007	Ben@MoreTechno...	10	41000.00	10	O'Haire	Terry	2004-10-...	2	3	NULL	2	222 Seco...	Boston	MA
7	008	www.@-Last-U-Ca...	7	25000.00	7	Lonning	David	2000-01-...	1	11	On Lea...	1	111 First ST	Seattle	WA
8	009	Thank you @.com	11	21500.00	11	Smith	Sally	1989-04-...	1	NULL	NULL	1	111 First ST	Seattle	WA
9	010	Call Mom @Com	5	7500.00	5	Bender	Eric	2007-05-...	1	11	NULL	1	111 First ST	Seattle	WA

Figure 2.18 An INNER JOIN on LocationID adds the Location table to the Grant-Employee join.

## Chapter 2. Query Options

Now specify just the three fields we want to see in our report. Waiting to narrow your field selection list until all needed tables have been joined in the FROM clause is a best practice for SQL Pros (see Figure 2.19).



```
SELECT GrantName, Amount, City
FROM [Grant] as gr INNER JOIN Employee as em
ON gr.EmpID = em.EmpID
INNER JOIN Location loc
ON em.LocationID = loc.LocationID
```

	GrantName	Amount	City
1	92 Purr_Scents %% team	4750.00	Seattle
2	K_Land fund trust	15750.00	Seattle
3	Robert@BigStarBank.com	18100.00	Seattle
4	BIG 6's Foundation%	21000.00	Seattle
5	TALTA_Kishan International	18100.00	Boston
6	Ben@MoreTechnology.com	41000.00	Boston
7	www.@-Last-U-Can-Help.com	25000.00	Seattle
8	Thank you @.com	21500.00	Seattle
9	Call Mom @Com	7500.00	Seattle

**Figure 2.19** Make specifying your field select list your last step when working with joins.

The first thing SQL Server evaluates in a query is the FROM clause, so it makes sense to first completely build that. When you try specifying the fields too early, you run the risk of getting column ambiguity errors which waste time and are frustrating. Build all your joins and see them successfully run with “SELECT \*” before narrowing down the fields you want to display in your result.

We now see the GrantName, Amount, and City fields in our report (see Figure 2.19). Open a new query window and look at all fields from just the Grant and Location tables (SELECT \* FROM [Grant], SELECT \* FROM Location). Notice these two tables do not have a common field. We cannot directly join these two tables. We were able to connect them indirectly because each table relates to the Employee table.

Could we add a fourth table here? Absolutely. Given our curiosity about Terry O’Haire, we could have gone an additional step to logically link to the PayRates table using EmpID. But for the moment we will focus on three table queries, since those pave the way to our next major topic: Many-To-Many Relationships.

## Lab 2.2: Three Table Query

**Lab Prep:** Before you can begin the lab you must run the SQLQueriesChapter2.2Setup.sql script. It is recommended that you view the lab video instructions found in Lab2.2\_ThreeTableQuery.wmv, available at [www.Joe2Pros.com](http://www.Joe2Pros.com).

**Skill Check 1:** Show all the city names and rates of pay for each Employee in those cities. You will need to join the Location, Employee, and PayRates table. Show the City field from the Location table. Include FirstName and LastName from the Employee table and all fields from the PayRates table. When you're done, your result should resemble Figure 2.20 (below).

	city	firstname	lastname	EmpID	YearlySalary	MonthlySalary	HourlyRate
1	Seattle	Alex	Adams	1	76000.00	NULL	NULL
2	Seattle	Barry	Brown	2	79000.00	NULL	NULL
3	Boston	Lee	Osako	3	NULL	NULL	45.00
4	Seattle	David	Kennson	4	NULL	6500.00	NULL
5	Seattle	Eric	Bender	5	NULL	5800.00	NULL
6	Spokane	Lisa	Kendall	6	5200.00	NULL	NULL
7	Seattle	David	Lonning	7	NULL	6100.00	NULL
8	Boston	James	Newton	9	NULL	NULL	18.00
9	Boston	Terry	O'Haire	10	NULL	NULL	17.00
10	Seattle	Sally	Smith	11	115000.00	NULL	NULL
11	Spokane	Barbara	O'Neil	12	NULL	NULL	21.00

**Figure 2.20** Joining the Location, Employee, and PayRates tables.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab2.2\_ThreeTableQuery.sql

## Three Table Query - Points to Ponder

1. In SQL Server you can join up to 256 tables in a single query.
2. The steps for joining three (or more) tables are the same as joining two tables. First – join all tables and confirm they are working properly using a “SELECT \*” statement. Narrow down the SELECT list to include the specific fields you need only after all tables have been joined.
3. While table aliasing saves keystrokes on two table joins, it greatly helps readability when joining three or more tables.

## Many-to-Many Relationships

How many kids do you have? The question implies you can have many. We also know you can have just one or even zero children. Asking a child “How many parents do you have?” again would give you different results. The relationship between parents and children goes beyond a single join on one ID field.

“Joan” is the soccer mother of the “Jay” family. Her two active children are named “Joey” and “Janet.” Joan picks up her children from school each day. She is also authorized to get them from school if they become sick or injured. Since Joan is a parent and she could have more than one child, we need to be able to handle her authorization to be the guardian for many children.



**Figure 2.21** Parents can have many children and children can have many parents

The reason you don’t list a ParentID in the child table is because most children have more than one ParentID. Which parent would you pick?

Let’s say Joey gets sick and the school nurse calls home. Joan is out of town so Joey’s father Ed shows up to get him. The problem is there is only one ParentID listed in the child table and it refers to Joan. The Parent and Child tables in this example don’t have an efficient way to join directly.

We need a new process to map this complex relationship. Sometimes tables exist for the sole purpose of allowing indirect relationships between tables. Common terms for these type of tables are Mapping tables, Bridge tables, or Junction tables.



**Figure 2.22** Joey has Ed as his Father

## Chapter 2. Query Options

Using the example above, a school might handle the Jay family and many other families with the design you see here (Figure 2.23).

dbo.Parent		dbo.ParentChild		dbo.Child	
P_ID	P_Name	P_ID	C_ID	C_ID	C_Name
1001	Joan	1001	2001	2001	Janet
1002	Sue	1001	2002	2002	Joey
1003	Sara	1002	2003	2003	Kayla
1004	Ed	1003	2004	2004	Robbie
1005	Jeff	1003	2005	2005	Andy
		1004	2001		
		1004	2002		
		1005	2003		

**Figure 2.23** ParentChild handles the many-to-many relationship between parents and children.

The Bridge table tells us that P\_ID of 1001 has two children of Janet and Joey.

dbo.Parent		dbo.ParentChild		dbo.Child	
P_ID	P_Name	P_ID	C_ID	C_ID	C_Name
1001	Joan	1001	2001	2001	Janet
1002	Sue	1001	2002	2002	Joey
1003	Sara	1002	2003	2003	Kayla
1004	Ed	1003	2004	2004	Robbie
1005	Jeff	1003	2005	2005	Andy

**Figure 2.24** Looking at the ParentID of 1001, the table dbo.ParentChild tells us Joan's children are Janet and Joey.

It also tells us that Joey has two parents of Joan and Ed. One parent can have many children and one child can have many parents (see Figure 2.25).

dbo.Parent		dbo.ParentChild		dbo.Child	
P_ID	P_Name	P_ID	C_ID	C_ID	C_Name
1001	Joan	1001	2001	2001	Janet
1002	Sue	1001	2002	2002	Joey
1003	Sara	1002	2003	2003	Kayla
1004	Ed	1003	2004	2004	Robbie
1005	Jeff	1003	2005	2005	Andy
		1004	2001		
		1004	2002		
		1005	2003		

**Figure 2.25** You can see that Joey has the parents Joan and Ed.

## Invoicing Systems

These JProCo tables use a many-to-many relationship to show which products were ordered on which invoices, the same way a school relates parents to children. A Product can appear on many invoices and an invoice can have many products. The relationship between Products and Invoices is known as a many-to-many relationship(see Figure 2.26).

dbo.Product		dbo.InvoiceDetail		dbo.Invoice	
P_ID	P_Name	P_ID	Inv_ID	Inv_ID	Inv_Date
1	Toy Car	1	5631	5631	1/1/2009
2	Furchee	2	5631	5632	3/5/2009
3	Timbot	2	5632	5633	7/15/2009
4	Go-Duck	5	5632	5634	9/22/2009
5	Pet Mock	1	5633		
		4	5633		
		4	5634		
		1	5634		

**Figure 2.26** A many-to-many relationship exists between Products and Invoices.

Look at Figure 2.26 above. If you were to ask yourself “What Products are on Invoice 5631?” or “Go-Duck was ordered on how many invoices?” you could resolve this query. Figure 2.27 shows us how to resolve both mappings

dbo.Product		dbo.InvoiceDetail		dbo.Invoice	
P_ID	P_Name	P_ID	Inv_ID	Inv_ID	Inv_Date
1	Toy Car	1	5631	5631	1/1/2009
2	Furchee	2	5631	5632	3/5/2009
3	Timbot	2	5632	5633	7/15/2009
4	Go-Duck	5	5632	5634	9/22/2009
5	Pet Mock	1	5633		
		4	5633		
		4	5634		
		1	5634		

**Figure 2.27** We can see Go-Duck was ordered twice. We also see that Invoice 5631 has a Toy Car and a Furchee.

Now we’ll take a look at many-to-many relationships between JProCo’s sales invoices and products (see Figure 2.28).

dbo.SalesInvoice						dbo.SalesInvoiceDetail					
InvoiceID	OrderDate	PaidDate	CustomerID	Comment		InvoiceDetailID	InvoiceID	ProductID	Quantity	UnitDiscount	
1	1	2006-01-03...	2006-01-11...	472	NULL	1	1	1	76	2	0.00
2	2	2006-01-04...	2006-02-01...	388	NULL	2	2	1	77	3	0.00
3	3	2006-01-04...	2006-02-14...	279	NULL	3	3	1	78	6	0.00
4	4	2006-01-04...	2006-02-08...	309	NULL	4	4	1	71	5	0.00
5	5	2006-01-05...	2006-02-10...	757	NULL	5	5	1	72	4	0.00
6	6	2006-01-06...	2006-01-28...	493	NULL	6	6	2	73	2	0.00
7	7	2006-01-08...	2006-02-05...	209	NULL	7	7	3	74	3	0.00
8	8	2006-01-08...	2006-02-17...	649	NULL	8	8	4	14	3	0.00
9	9	2006-01-08...	2006-01-27...	597	NULL	9	9	5	16	1	0.00
10	10	2006-01-09...	2006-02-20...	736	NULL	10	10	5	9	5	0.00
11	11	2006-01-10...	2006-01-18...	329	NULL	11	11	5	12	3	0.00
12	12	2006-01-11...	2006-01-22...	52	NULL	12	12	5	11	6	0.00

dbo.CurrentProducts						
ProductID	ProductName				ToBeDeleted	Category
1	1	Underwater Tour 1 Day West Coast	61.483	2006-08-11 13:33:09.957	0	No-Stay
2	2	Underwater Tour 2 Days West Coast	110.6894	2007-10-03 23:43:22.813	0	Overnight-Stay
3	3	Underwater Tour 3 Days West Coast	184.449	2009-05-09 16:07:49.900	0	Medium-Stay
4	4	Underwater Tour 5 Days West Coast	245.932	2006-03-04 04:59:06.600	0	Medium-Stay
5	5	Underwater Tour 1 Week West Coast	307.415	2001-07-18 19:20:11.400	0	LongTerm-Stay
6	6	Underwater Tour 2 Weeks West Coast	553.347	2008-06-30 20:40:38.760	0	LongTerm-Stay
7	7	Underwater Tour 1 Day East Coast	80.859	2007-04-07 08:25:43.233	0	No-Stay
8	8	Underwater Tour 2 Days East Coast	145.5462	2005-06-11 09:52:12.910	0	Overnight-Stay
9	9	Underwater Tour 3 Days East Coast	242.577	2003-04-01 04:59:05.850	0	Medium-Stay
10	10	Underwater Tour 5 Days East Coast	323.436	2005-04-13 04:34:35.027	0	Medium-Stay
11	11	Underwater Tour 1 Week East Coast	404.295	2004-09-09 13:15:33.183	0	LongTerm-Stay
12	12	Underwater Tour 2 Weeks East Coast	727.731	2008-11-25 19:59:47.407	0	LongTerm-Stay

Figure 2.28 Many-to-many relationships in JProCo’s sales invoices and products.

Here is an example of JProCo’s sales invoices mapping to a bridge table (SalesInvoiceDetail) in order to map over to the CurrentProducts table. The CurrentProducts table gives us all the detail of the current products that have been ordered.

Let’s take a look at SalesInvoice 5 (see Figure 2.28). It looks like many products were ordered on that one invoice (Products 9, 11, 12, and 16). To see what those products are, we would look over to the CurrentProducts table. We see Product 9 is an Underwater Tour 3 Days East Coast. Product 11 is an Underwater Tour 1 Week East Coast, and so forth.

So a SalesInvoice can have many products, and products can be ordered on multiple sales invoices.

## Lab 2.3: Many-To-Many Relationships

**Lab Prep:** Before you can begin the lab you must run the SQLQueriesChapter2.3Setup.sql script. It is recommended that you view the lab video instructions found in Lab2.3\_ManyToManyRelationships.wmv, available at [www.Joe2Pros.com](http://www.Joe2Pros.com).

**Skill Check 1:** Write a query that shows all the invoices ordered by customer 490. Show all fields from both tables (SalesInvoice and SalesInvoiceDetail). When you're done your result should resemble the figure you see here.

	InvoiceDetailID	InvoiceID	ProductID	Quantity	UnitDiscount	InvoiceID	OrderDate	PaidDate	CustomerID	Comment
1	5057	1285	64	4	0.00	1285	2008-11-04 ...	2008-12-23 ...	490	NULL
2	5568	1459	70	2	0.00	1459	2009-03-16 ...	2009-04-27 ...	490	NULL
3	6700	1804	49	1	0.00	1804	2009-12-25 ...	2010-01-24 ...	490	NULL

Figure 2.29 Skill Check 1 shows the Customer who ordered each invoice.

**Skill Check 2:** Write a query that combines SalesInvoice, SalesInvoiceDetail and CurrentProducts. Show the following fields:

- SalesInvoice.CustomerID
- SalesInvoice.InvoiceID
- SalesInvoice.OrderDate
- SalesInvoiceDetail.Quantity
- CurrentProducts.ProductName
- CurrentProducts.RetailPrice

When you are done your result should resemble the figure you see here.

```
SELECT si.CustomerID, si.InvoiceID, si.OrderDate,
```

	CustomerID	InvoiceID	OrderDate	quantity	ProductName	RetailPrice
1	597	9	2006-01-08 21:46:03.093	5	Underwater Tour 1 Day East Coast	80.859
2	736	10	2006-01-09 20:33:07.380	2	Underwater Tour 1 Day East Coast	80.859
3	47	15	2006-01-13 10:40:06.230	3	Underwater Tour 1 Day East Coast	80.859
4	251	19	2006-01-16 18:00:24.947	4	Underwater Tour 1 Day East Coast	80.859
5	529	20	2006-01-17 06:36:28.483	5	Underwater Tour 1 Day East Coast	80.859
6	151	22	2006-01-17 23:41:13.490	5	Underwater Tour 1 Day East Coast	80.859
7	191	23	2006-01-18 04:55:24.700	5	Underwater Tour 1 Day East Coast	80.859
8	231	25	2006-01-18 19:01:45.203	3	Underwater Tour 1 Day East Coast	80.859

JProCo 00:00:00 6960 rows

Figure 2.30 Skill Check 2 shows all the product details for each InvoiceID.

**Answer Code:** The T-SQL code to this lab can be found in the downloadable files in a file named Lab2.3\_Many-to-Many\_Relationships.sql

## Many-to-Many - Points to Ponder

1. Databases often contain tables which exist for the sole purpose of allowing indirect relationships between tables.
2. These tables are known as mapping tables, bridge tables, or junction tables.

## Chapter Two - Review Quiz

1.) You have a table name CurrentProducts and need to display just the [ProductName] and [Category]. You want the highest [RetailPrice] listed first and the lowest price listed last. Which query should you use?

- O a. 

```
SELECT ProductName, Category
FROM CurrentProducts
ORDER BY ProductName, Category DESC
```
- O b. 

```
SELECT *
FROM CurrentProducts
ORDER BY ProductName, Category DESC
```
- O c. 

```
SELECT ProductName, Category
FROM CurrentProducts
ORDER BY RetailPrice ASC
```
- O d. 

```
SELECT ProductName, Category
FROM CurrentProducts
ORDER BY RetailPrice DESC
```

2.) You have a table named Employee and need to display just the [FirstName], [LastName], and [HireDate]. You want the most recent hire date listed first. Which query will achieve this goal?

- O a. 

```
SELECT FirstName, LastName, HireDate
FROM Employee
ORDER BY FirstName DESC
```
- O b. 

```
SELECT FirstName, LastName, HireDate
FROM Employee
ORDER BY LastName DESC
```
- O c. 

```
SELECT FirstName, LastName, HireDate
FROM Employee
ORDER BY HireDate DESC
```
- O d. 

```
SELECT FirstName, LastName, HireDate
FROM Employee
ORDER BY FirstName ASC
```
- O e. 

```
SELECT FirstName, LastName, HireDate
FROM Employee
ORDER BY LastName ASC
```

## Chapter 2. Query Options

3.) You need a report of all invoices numbers that have an InvoiceID of over 500. Each invoice should be listed each time it was a customer placed an order. The report should show the FirstName and LastName of the customer who placed the order. This report will be sorted alphabetically by ProductName and lists all the dates each product was ordered. Which query should you use?

- O a. `SELECT cu.Firstname, cu.LastName, si.InvoiceID, si.OrderDate  
FROM Customer cu INNER JOIN SalesInvoice si  
ON cu.CustomerID = si.customerID  
WHERE InvoiceID < 500  
ORDER BY ProductName`
- O b. `SELECT cu.Firstname, cu.LastName, si.InvoiceID, si.OrderDate  
FROM Customer cu INNER JOIN SalesInvoice si  
ON cu.CustomerID = si.customerID  
WHERE InvoiceID > 500  
ORDER BY ProductName`
- O c. `SELECT cu.Firstname, cu.LastName, si.InvoiceID, si.OrderDate  
FROM Customer cu INNER JOIN SalesInvoice si  
ON cu.CustomerID = si.customerID  
WHERE InvoiceID < 500  
ORDER BY ProductName DESC`
- O d. `SELECT cu.Firstname, cu.LastName, si.InvoiceID, si.OrderDate  
FROM Customer cu INNER JOIN SalesInvoice si  
ON cu.CustomerID = si.customerID  
WHERE InvoiceID > 500  
ORDER BY ProductName DESC`

4.) Your manager wants to see all employees in alphabetical order for each state. She has asked you to sort by State, LastName, and FirstName columns. Without creating any additional tables how can you view this report?

- O a. Specify State, LastName, and FirstName in the ORDER command.
- O b. Create a file format that has an export operation.
- O c. Specify State, LastName, and FirstName in the ORDER BY clause.
- O d. Copy the data into a new table that has a clustered index set on State, LastName, and FirstName.

5.) You have tables named `dbo.SalesInvoice` and `dbo.SalesInvoiceDetail`. `CustomerID` is located in the `SalesInvoice` table and `InvoiceID` is located in both tables. You have been told to show the discount amounts from the `SalesInvoiceDetail` table that correspond to the sales of a specific `CustomerID` of 490. Which Transact SQL statement should you use?

- O a. `SELECT CustomerID, DiscountAmt  
FROM dbo.SalesInvoiceDetail sd  
INNER JOIN dbo.SalesInvoice si  
ON sd.InvoiceID= si.InvoiceID  
WHERE si.CustomerID= 490`
- O b. `SELECT CustomerID, DiscountAmt  
FROM dbo.SalesInvoiceDetail sd  
WHERE si.CustomerID= 490`
- O c. `SELECT CustomerID, DiscountAmt  
FROM dbo.SalesInvoiceDetail sd  
WHERE EXISTS (dbo.SalesInvoice si  
ON sd.InvoiceID= si.InvoiceID  
WHERE si.CustomerID= 490)`

## Answer Key

1.) The code in (a) is sorting the result set by `ProductName` and `Category` instead of the `RetailPrice` as requested so it is incorrect. `SELECT *` would return every field rather than limiting to just the `ProductName` and `Category` as requested so (b) is incorrect too. To order a result set from highest to lowest you would use the `DESC` keyword not the `ASC` keyword so (c) is also incorrect. The code in (d) limits the field list to only `ProductName` and `Category` and sorts the result set by `RetailPrice` from highest to lowest using the `DESC` keyword making (d) the correct answer.

2.) Because the code in (a) and (d) are sorting the result set in `ORDER BY FirstName` instead of `HireDate` they are wrong. Sorting the results in `ORDER BY LastName` as in (b) and (e) makes them both wrong too. To list the result set in 'ORDER BY `HireDate`' from the most recent to the furthest in the past, use the `DESC` keyword because newer dates are greater in value than older dates which makes (c) the correct answer.

3.) Since using 'WHERE `InvoiceID < 500`' will return all records that have an `InvoiceID` of 499 or lower (a) and (c) are both incorrect. Because the code in (d)

is sorting the results in ORDER BY ProductName DESC they will be displayed in reverse alphabetical order making it wrong too. The right answer is (b) because it is filtering the records WHERE InvoiceID > 500 and is sorting the results in ORDER BY ProductName (by default ORDER BY uses ASC).

4.) 'ORDER' is not a command but part of the 'ORDER BY' clause so (a) is wrong. 'Creating a file format that has an export operation' has nothing to do with sorting a result set so (b) is wrong too. Because your manager specifically requested not creating any additional tables (d) is also incorrect. To meet the requirements 'Specify State, LastName, and FirstName in the ORDER BY clause' which makes (c) the correct answer.

5.) The code in (b) does not look at the SalesInvoice Table which contains the CustomerID field so it will return an error. The use of EXISTS with a subquery (see chapters 6 & 11) in the WHERE clause will either return every row in the SalesInvoiceDetail table or none of them, regardless of the CustomerID, so (c) is also incorrect. Because 'dbo.SalesInvoiceDetail sd INNER JOIN dbo.SalesInvoice si ON sd.InvoiceID = si.InvoiceID' will return only records that have a match from both tables and 'WHERE si.CustomerID = 490' will filter the results to only display records having a CustomerID of 490, (a) is the correct answer.

## Bug Catcher Game

To play the Bug Catcher game run the SQLQueriesBugCatcherCh2.pps from the BugCatcher folder of the companion files. You can obtain these files from [www.Joes2Pros.com](http://www.Joes2Pros.com) or by ordering the Companion CD.