Professional Expertise Distilled

OCA Oracle Database 11g: SQL Fundamentals I: A Real-World Certification Guide

Ace the 1Z0-051 SQL Fundamentals I exam, and become a successful DBA by learning how SQL concepts work in the real world

Steve Ries

[PACKT] PUBLISHING enterprise professional expertise distilled

OCA Oracle Database 11g: SQL Fundamentals I: A Real-World Certification Guide

Ace the 1Z0-051 SQL Fundamentals I exam, and become a successful DBA by learning how SQL concepts work in the real world

Steve Ries

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

OCA Oracle Database 11g: SQL Fundamentals I: A Real-World Certification Guide

First published: November 2011

Production Reference: 1171111

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-364-7

www.packtpub.com

Cover Image by Sandeep Babu (sandyjb@gmail.com)

Credits

Author
Steve Ries

Reviewers
Dmitri Levin
Claire Rajan

Acquisition Editor
Amey Kanse

Development Editors
Pallavi Iyengar
Meeta Rajani

Technical Editors
Apoorva Bolar
Arun Nadar
Naheed Shaikh

Copy Editor
Brandt D'Mello

Project Coordinator
Leena Purkait

Proofreader
Jonathan Todd

Indexer
Monica Ajmera Mehta

Graphics
Valentina D'silva
Manu Joseph

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Steve Ries has been an Oracle DBA for 15 years, specializing in all aspects of database administration, including security, performance tuning, and backup and recovery. He is a specialist in Oracle Real Application Clusters (RAC) and has administered Oracle clustered environments in every version of Oracle since the creation of Oracle Parallel Server. He holds five Oracle certifications as well as the Security+ certification. He currently consults for the Dept of Defense, U.S. Marine Corps, and holds a high-level security clearance. Additionally, Steve has been an adjunct instructor of Oracle technologies at Johnson County Community College for eight years where he teaches classes that prepare students for the Oracle certification exams. He was also a speaker at the 2011 Oracle Open World conference. Steve is an award-winning technical paper writer and the creator of the alt.oracle blog.

I would like to thank Gary Hayes, Carol Ross, Matt Sams, Pete Scalzi, Angela Morten, Joe Duvall, Sandee Vandenboom, Karen Buck, Gary Deardorff, and Chad Fletcher for their support and technical advice during the writing of this book. I would also like to thank Debbie Rulo, Keith Krieger, and the staff at the Center for Business at Johnson County Community College for their support. Finally, I would like to thank my wife Dee and daughter Faith for their love, personal support, and patience.

About the Reviewers

Dmitri Levin has been working as a database administrator for more than 15 years. His areas of interest include database design, database replication, and database performance tuning. Dmitri has spoken at several national and international conferences. He is currently Sr. Database Architect and Administrator at Broder Bros Co. Dmitri has an MS in Mathematics from St. Petersburg University, Russia, and is an Oracle Database 11g Certified Associate.

Claire Rajan is an Oracle instructor, author, and database consultant. She currently instructs at the American Career Institute, MD, where she teaches Oracle Database administration. She has over 15 years of experience managing Oracle databases and teaching Oracle-related topics. She has created and maintains the popular website www.oraclecoach.com. The website provides a host of articles, videos, and technical resources for both beginners and advanced learners. She has authored the book Oracle 10g Database Administrator II: Backup/Recovery and Network Administration, published by Cengage Learning. She holds certifications in all major Oracle releases: 7.x, 8, 8i, 9i, 10g, and 11g. She can be found on Linkedin (`http://www.linkedin.com/in/clairerajan`). She can be reached at `cr@oraclecoach.com`.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface **1**

Chapter 1: SQL and Relational Databases **7**

Relational Database Management Systems **8**

Flat file databases 8

Limitations of the flat file paradigm 9

Normalization 10

The relational approach 13

Bringing it into the Oracle world 16

Tables and their structure 16

Structured Query Language **18**

A language for relational databases 18

Commonly-used SQL tools 20

SQL*Plus 20

TOAD 21

DBArtisan 22

SQL Worksheet (Enterprise Manager) 23

PL/SQL Developer 24

Oracle SQL Developer 24

Working with SQL **25**

Introducing the Companylink database 25

An introduction to Oracle SQL Developer **27**

Setting up SQL Developer 27

Getting around in SQL Developer 31

Summary **34**

Test your knowledge **34**

Chapter 2: SQL SELECT Statements **37**

The purpose and syntax of SQL **38**

The syntax of SQL 38

Case sensitivity 39

The use of whitespace 40
Statement terminators 41
Retrieving data with SELECT statements 42
Projecting columns in a SELECT statement 42
Selecting a single column from a table 43
Selecting multiple columns from a table 44
Selecting all columns from a table 46
Displaying the structure of a table using DESCRIBE 48
Using aliases to format output of SELECT statements 50
Using arithmetic operators with SELECT 53
The DUAL table and the use of string literals 54
Mathematical operators with SELECT 57
The meaning of nothing 60
Using DISTINCT to display unique values 62
Concatenating values in SELECT statements 65
Summary 69
Certification objectives covered 69
Test your knowledge 70
Chapter 3: Using Conditional Statements 75
Implementing selectivity using the WHERE clause 76
Understanding the concept of selectivity 76
Understanding the syntax of the WHERE clause 76
Using conditions in WHERE clauses 79
Using equality conditions 79
Implementing non-equality conditions 82
Examining conditions with multiple values 86
Constructing range conditions using the BETWEEN clause 86
Using the IN clause to create set conditions 89
Pattern-matching conditions using the LIKE clause 91
Understanding Boolean conditions in the WHERE clause 94
Examining the Boolean OR operator 95
Understanding the Boolean AND operator 97
The Boolean NOT operator 98
Using ampersand substitution with runtime conditions 101
Sorting data 102
Understanding the concepts of sorting data 102
Sorting data using the ORDER BY clause 103
Changing sort order using DESC and ASC 104
Secondary sorts 106
Summary 110
Certification objectives covered 110

Test Your Knowledge **111**

Chapter 4: Data Manipulation with DML **115**

Persistent storage and the CRUD model **115**
Understanding the principles of persistent storage 116
Understanding the CRUD model and DML 117
Creating data with INSERT **118**
Examining the syntax of the INSERT statement 118
Using single table inserts 119
Inserts using positional notation 119
Inserts using named column notation 121
Inserts using NULL values 122
Multi-row inserts 124
Conditional Inserts—INSERT...WHEN 125
Modifying data with UPDATE **128**
Understanding the purpose and syntax of the UPDATE statement 128
Writing single-column UPDATE statements 128
Multi-column UPDATE statements 131
Removing data with DELETE **132**
The purpose and syntax of the DELETE statement 133
Deleting rows by condition 133
Deleting rows without a limiting condition 135
Removing data unconditionally with TRUNCATE 136
Transaction control **138**
Transactions and the ACID test 139
Completing transactions with COMMIT 140
Undoing transactions with ROLLBACK 142
DELETE and TRUNCATE revisited 146
Recognizing errors **146**
Summary **149**
Certification objectives covered 149
Test your knowledge **149**

Chapter 5: Combining Data from Multiple Tables **155**

Understanding the principles of joining tables **155**
Accessing data from multiple tables 156
The ANSI standard versus Oracle proprietary syntax 158
Using ANSI standard joins **159**
Understanding the structure and syntax of ANSI join statements 159
Examining ambiguous Cartesian joins 160
Using equi joins—joins based on equivalence 162
Implementing two table joins with a table-dot notation 162
Using two table joins with alias notation 165

Understanding the row inclusiveness of outer joins 166
Retrieving data from multiple tables using n-1 join conditions 171
Working with less commonly-used joins—non-equi joins and self-joins 176
Understanding Oracle join syntax 178
Using Cartesian joins with Cross join 178
Joining columns ambiguously using NATURAL JOIN 180
Joining on explicit columns with JOIN USING 184
Constructing fully-specified joins using JOIN ON 186
Writing n-1 join conditions using Oracle syntax 189
Creating multi-table natural joins 190
Building multi-table joins with JOIN USING 190
Summary 191
Certification objectives covered 192
Test your knowledge 192
Chapter 6: Row Level Data Transformation 197
Understanding functions and their use 197
Comprehending the principles of functions 198
Using single-row functions for data transformation 198
Understanding String functions 199
Using case conversion functions 199
UPPER() 200
LOWER() 202
INITCAP() 202
Writing SQL with String manipulation functions 203
LENGTH() 204
Padding characters with LPAD() and RPAD() 206
RTRIM() and LTRIM() 208
CONCAT() 208
SUBSTR() 209
INSTR() 212
Exploring nested functions 214
Handling DATE functions 217
Distinguishing SYSDATE and CURRENT_TIMESTAMP 217
Utilizing datatype conversion functions 219
Using date to character conversion with TO_CHAR 219
Converting characters to dates with TO_DATE() 223
Converting numbers using TO_NUMBER() 224
Using arithmetic functions 227
ROUND() 227
TRUNC() 229
Using ROUND() and TRUNC() with dates 229
MOD() 230
Understanding date arithmetic functions 231

MONTHS_BETWEEN() 232
ADD_MONTHS() 233
Examining functions that execute conditional retrieval 233
NVL() 234
NVL2() 235
DECODE() 236
Summary 237
Certification Objectives Covered 237
Test your knowledge 238
Chapter 7: Aggregate Data Transformation 243
Understanding multi-row functions 244
Examining the principles of grouping data 244
Using multi-row functions in SQL 244
COUNT() 245
MIN() and MAX() 248
SUM() 250
AVG() 251
Grouping data 252
Grouping data with GROUP BY 254
Avoiding pitfalls when using GROUP BY 256
Extending the GROUP BY function 260
Using statistical functions 262
STDDEV() 262
VARIANCE() 263
Performing row group exclusion with the HAVING clause 263
Putting it all together 266
Certification objectives covered 267
Summary 267
Test your knowledge 267
Chapter 8: Combining Queries 271
Understanding the principles of subqueries 271
Accessing data from multiple tables 272
Solving problems with subqueries 272
Examining different types of subqueries 274
Using scalar subqueries 274
Using scalar subqueries with WHERE clauses 275
Using scalar subqueries with HAVING clauses 277
Using scalar subqueries with SELECT clauses 278
Processing multiple rows with multi-row subqueries 280
Using IN with multi-row subqueries 280
Using ANY and ALL with multi-row subqueries 282
Using multi-row subqueries with HAVING clauses 286
Using correlated subqueries 287

Using multi-column subqueries 289
Using multi-column subqueries with WHERE clauses 290
Multi-column subqueries with the FROM clause 291
Investigating further rules for subqueries 292
Nesting subqueries 292
Using subqueries with NULL values 294
Using set operators within SQL 296
Principles of set theory 296
Comparing set theory and relational theory 297
Understanding set operators in SQL 298
Using the INTERSECT set operator 298
Using the MINUS set operator 299
Using the UNION and UNION ALL set operators 300
Summary 302
Certification objectives covered 302
Test your knowledge 303
Chapter 9: Creating Tables 307
Introducing Data Definition Language 307
Understanding the purpose of DDL 308
Examining Oracle's schema-based approach 308
Understanding the structure of tables and datatypes 309
CHAR 310
VARCHAR2 311
NUMBER 312
DATE 313
Other datatypes 313
Using the CREATE TABLE Statement 314
Understanding the rules of table and column naming 314
Creating tables 315
Avoiding datatype errors 318
Avoiding character datatype errors 318
Avoiding numeric datatype errors 322
Copying tables using CTAS 326
Modifying tables with ALTER TABLE 329
Adding columns to a table 329
Changing column characteristics using ALTER TABLE... MODIFY 332
Removing columns using ALTER TABLE... DROP COLUMN 335
Removing tables with DROP TABLE 337
Using database constraints 338
Understanding the principles of data integrity 338
Enforcing data integrity using database constraints 339
NOT NULL 339
PRIMARY KEY 341
Natural versus synthetic 345

FOREIGN KEY 345
Deleting values with referential integrity 347
UNIQUE 348
CHECK 348
Extending the Companylink Data Model 349
Adding constraints to Companylink tables 349
Adding referential integrity 350
Adding a NOT NULL constraint 352
Adding a CHECK constraint 352
Adding tables to the Companylink model 353
Summary 356
Certification objectives covered 356
Test your knowledge 356
Chapter 10: Creating Other Database Objects 361
Using indexes to increase performance 361
Scanning tables 362
Understanding the Oracle ROWID 362
Examining B-tree indexes 364
Creating B-tree indexes 366
Using composite B-tree indexes 368
Working with bitmap indexes 369
Understanding the concept of cardinality 369
Examining the structure of bitmap indexes 370
Creating a bitmap index 371
Working with function-based indexes 372
Modifying and dropping indexes 374
Working with views 375
Creating a view 375
Creating selective views 377
Distinguishing simple and complex views 378
Configuring other view options 381
Changing or removing a view 381
Using sequences 382
Using sequences to generate primary keys 382
Object naming using synonyms 386
Schema naming 387
Using synonyms for alternative naming 388
Creating private synonyms 388
Creating public synonyms 391
Summary 392
Certification objectives covered 392
Test your knowledge 392

Chapter 11: SQL in Application Development 397
Using SQL with other languages 398
Why SQL is paired with other languages 398
Using SQL with PL/SQL 398
Using SQL with Perl 401
Using SQL with Python 403
Using SQL with Java 404
Understanding the Oracle optimizer 405
Rule-based versus cost-based optimization 406
Gathering optimizer statistics 406
Viewing an execution plan with EXPLAIN PLAN 408
Advanced SQL statements 411
Exam preparation 415
Helpful exam hints 415
A recommended strategy for preparation 417
Summary 417
Appendix A: Companylink Table Reference 419
The Companylink data model 419
ADDRESS 419
AWARD 420
BLOG 420
BRANCH 420
DIVISION 420
EMAIL 421
EMPLOYEE 421
EMPLOYEE_AWARD 421
MESSAGE 422
PROJECT 422
WEBSITE 422
Appendix B: Getting Started with APEX 423
Oracle Application Express 423
What is APEX? 423
Signing up for APEX 424
Using APEX 428
Index 433

Preface

There's never been a time in the Information Technology industry where professional certifications have been more important. Because of the specialized nature of technological careers today, certifications are considered by some to be just as important as technological degrees. This focus on certifications has led to the rise of an entire industry around books that assist readers in preparing for various certification tests. In the author's opinion, many, if not most, of these books make a lot of assumptions as to the prior knowledge of the reader and serve more as reference material than a cohesive learning experience.

In my role as an instructor of Oracle technologies, I have noticed a shift in the types of people seeking to learn Oracle. In the past several years, more and more students are seeking to break into an Oracle career path with little or no experience. Whether they come from backgrounds in business analysis, project management, or other non-database technical areas, these students need to be able to learn Oracle from the ground up. When instructing these types of students, I cannot make assumptions as to the knowledge they bring with them. We must start at the beginning and work our way to certification level knowledge. To accomplish this goal in class, the accompanying textbook must be designed in the same way.

Similarly, many certification books today serve only as exam cram books that neglect an application to real world scenarios. Readers of these types of books may find themselves with a certification, yet possess no way to apply the knowledge in their first job.

My goal in writing this book is to address both of these problems. This book attempts to begin at the foundation and continue to the knowledge of the subject required for the certification exam, using real world examples and tips along the way. This book is heavily example-oriented and is intended to serve as step-by-step instruction instead of reference material. In essence, I attempt to bring the classroom experience to the reader, using examples, real world tips, and end-of-the-chapter review. This book is written to be read cover to cover, with the reader completing the examples and questions as they go. Using this process, it is my hope that readers can truly begin at the beginning, regardless of previous experience, and learn SQL in a relevant way that will serve them in their pursuit of an Oracle certification as well as an Oracle career path.

The Oracle Database 11g: SQL Fundamentals I exam is the first stepping stone in earning the Oracle Certified Associate Certification for Oracle Database 11g. The SQL programming language is used in every major relational database today, and understanding the real-world application of it is the key to becoming a successful DBA.

This book gives you the essential real-world skills to master relational data manipulation with Oracle SQL and prepares you to become an Oracle Certified Associate. Beginners are introduced to concepts in a logical manner while practitioners can use it as a reference to jump to relevant concepts directly.

We begin with the essentials of why databases are important in today's information technology world and how they work. We continue by explaining the concepts of querying and modifying data in Oracle using a range of techniques, including data projection, selection, creation, joins, sub-queries, and functions. Finally, we learn to create and manipulate database objects and to use them in the same way as today's expert SQL programmers.

This book prepares you to master the fundamentals of the SQL programming language using an example-driven approach that is easy to understand.

This definitive certification guide provides a disciplined approach to be adopted for successfully clearing the 1Z0-051 SQL Fundamentals I exam, which is the first stepping stone towards attaining the OCA on Oracle Database 11g certification.

Each chapter contains ample practice questions at the end. A full-blown mock test is included for practice so you can test your knowledge and get a feel for the actual exam.

What this book covers

Chapter 1, *SQL and Relational Database,* examines the purpose and use of relational database management systems, including the use of entity relationship diagrams and the structure of tables. We then introduce Structured Query Language and the SQL Developer tool.

Chapter 2, *SQL SELECT Statements*, explores the most foundational SQL clause; the SELECT statement. We use SELECT statements for single and multi-column data retrieval and take a look at using SQL to do basic mathematical operations.

Chapter 3, *Using Conditional Statements,* examines the concept of data selection using the WHERE clause paired with conditions. In it, we construct selective statements using conditions of both equality and non-equality. We also use range and set conditions with the WHERE clause for further data selectivity. Finally, we examine the concept of sorting data using the ORDER BY clause.

Chapter 4, *Data Manipulation with DML*, explores the use of Data Manipulation Language to add, modify, and remove table data using INSERT, UPDATE, and DELETE statements. Lastly, we look at transaction control in SQL.

Chapter 5, *Combining Data from Multiple Tables*, examines the concept of combining data from multiple tables using various join statements. We accomplish this using both ANSI standard and Oracle syntax.

Chapter 6, *Row Level Data Transformation*, explores the concept of row level data transformation using single-row functions. We use these functions to transform date, character and numeric data.

Chapter 7, *Aggregate Data Transformation,* explores data transformation using functions, this time with multi-row functions. We combine these functions with the GROUP BY and HAVING statements to perform aggregate data transformation.

Chapter 8, *Combining Queries,* focuses on using several types of subqueries to combine data from multiple tables. We close the chapter by exploring set theory in Oracle and implement it using SQL set operators.

Chapter 9, *Creating Tables*, introduces the concept of Data Definition Language and how to use it to create database tables. We also use SQL to write database constraints that enforce business data rules.

Chapter 10, *Creating Other Database Objects,* examines the use of DDL statements to create some of the other common objects available to us in Oracle. We use these statements to create indexes, views, sequences, and synonyms.

Chapter 11, *Using SQL in Application Development,* examines how SQL is used in real-world programming languages such as PL/SQL, Perl, Python, and Java. We close by offering hints and strategies for taking the SQL certification exam.

Appendix A, Companylink Table Reference, a reference section describing the various tables used as examples in this book.

Appendix B, Getting Started with APEX, shows an alternative method for completing the examples in this book that does not require installing the Oracle database software.

Appendix C, *Test Your Knowledge*; you can download this appendix that contains answers to the *Test your knowledge* section in all the chapters at `http://www.packtpub.com/sites/default/files/downloads/testyourknowledge_answers.pdf`.

Mock practice test paper can be downloaded from `http://www.packtpub.com/sites/default/files/downloads/mock_test_paper.pdf`

What you need for this book

This book is heavily example-oriented. As such, it will be beneficial for the reader to download and install the Oracle database software as outlined in *Chapter 1*, *SQL and Relational Database*. The reader will also receive the greatest benefit by downloading and running the example code available from the Packt support website. No prior knowledge of programming or database concepts is required.

Who this book is for

This book is for anyone who needs the essential skills to pass the Oracle Database SQL Fundamentals I exam and use those skills in daily life as an SQL developer or database administrator.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text are shown as follows: "Simply unzip the `companylink.zip` file into a directory and double-click the `companylink_db.cmd` file."

A block of code is set as follows:

```
SELECT {column, column, …}
FROM {table};
```

Any command-line input or output is written as follows:

```
INSTR(column_expression, search_character, starting_position,
occurrence_number)
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In our example, data such as **Firstname**, **lastname**, **Address** and **Branch** name are the attributes of the **Employee** entity".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com` and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
SQL and Relational Databases

We live in a data-driven world. Think for a moment about all the data that exists about you, in computers around the world.

- Your name
- Birth date and information
- Your hobbies
- Purchases you've made
- The identity of your friends
- Your place of employment

The examples are endless. Next, multiply that amount of data by the number of people in the world. The result is a truly staggering amount of information. How is it possible that all this data can be organized and retrieved? In today's data-centric world, it is databases that make this possible. These **Relational Database Management Systems** (**RDBMS**) are primarily controlled by a programming language called **Structured Query Language** (**SQL**).

In this chapter, we will cover the following topics:

- Discussing the purpose of relational database management systems
- Understanding the use of the relational paradigm
- Examining the use of Entity Relationship Diagrams (ERDs)
- Looking at the structure of tables
- Introducing Structured Query Language (SQL)
- Reviewing commonly-used query tools
- Introducing the SQL Developer tool

Relational Database Management Systems

Imagine, for a moment, that you have the telephone books for the 20 largest cities in the U.S. I give you the following request: *Please find all the phone numbers for individuals named Rick Clark in the Greater Chicago area*. In order to satisfy the request, you simply do the following:

- Open the Chicago phone book
- Scan to the "C" section of names
- Find all individuals that match "Clark, Rick"
- Report back their phone numbers

Now imagine that I take each phone book, tear out all of the pages, and throw them into the air. I then proceed to shuffle the thousands of pages on the ground into a completely disorganized mess. Now I repeat the same request: *Please find all the phone numbers for individuals named Rick Clark in the Greater Chicago area*. How do you think you would do that? It would be nearly impossible. The data is all there, but it's completely disorganized. Finding the "Rick Clarks" of Chicago would involve individually examining each page to see if it satisfied the request—a very frustrating undertaking, to say the least.

This example underscores the importance of a database, or more accurately, a **Relational Database Management System(RDBMS)** Today's RDBMSs are what enable the storage, modification, and retrieval of massive amounts of data.

Flat file databases

When the devices that we know as computers first came into existence, they were primarily used for one thing—computation. Computers became useful entities because they were able to do numeric computation on an unprecedented scale. For example, one of the first computers, ENIAC, was designed (although not used) for the US Army to calculate artillery trajectories, a task made simpler through the use of complex sequences of mathematical calculations. As such, originally, computers were primarily a tool for mathematical and scientific research. Eventually, the use of computers began to penetrate the business market, where the company's data itself became just as important as computational speed. As the importance of this data grew, so the need for data storage and management grew as well, and the concept of a database was born.

The earliest databases were simple to envision. Most were simply large files that were similar in concept to a spreadsheet or **comma-separated values** (**CSV**) file. Data was stored as fields. A portion of these databases might look something like the following:

```
Susan, Bates, 123 State St, Somewhere, VA
Fred, Hartman, 234 Banner Rd, Anywhere, CA
Bill, Frankin, 345 Downtown Rd, Somewhere, MO
Emily, Thompson, 456 Uptown Rd, Somewhere, NY
```

In this example, the first field is determined by reading left to right until a delimiter, in this case a comma, is reached. This first field refers to the first name of the individual. Similarly, the next field is determined by reading from the first delimiter to the next. That second field refers to the last name of the individual. It continues in this manner until we have five fields—first name, last name, street address, city, and state. Each individual line or record in the file refers to the information for a distinct individual. Because this data is stored in a file, it is often referred to as a flat file database. To retrieve a certain piece of information, programs could be written that would scan through the records for the requested information. In this way, large amounts of data could be stored and retrieved in an orderly, programmatic way.

Limitations of the flat file paradigm

The flat file database system served well for many years. However, as time passed and the demands of businesses to retain more data increased, the flat file paradigm began to show some flaws.

In our previous example, our flat file is quite limited. It contains only five fields, representing five distinct pieces of information. If this flat file database contained the data for a real company, five distinct pieces of information would not even begin to suffice. A complete set of customer data might include addresses, phone numbers, information about what was ordered, when the order was placed, when the order was delivered, and so on. In short, as the need to retain more data increases, the number of fields grows. As the number of fields grows, our flat file database gets wider and wider. We should also consider the amount of data being stored. Our first example had four distinct records; not a very realistic amount for storing customer data. The number of records could actually number in thousands or even millions. Eventually, it is completely plausible that we could have a single flat file that is hundreds of fields wide and millions of records long. We could easily find that the speed with which our original data retrieval programs can retrieve the required data is decreasing at a rapid rate and is insufficient for our needs.

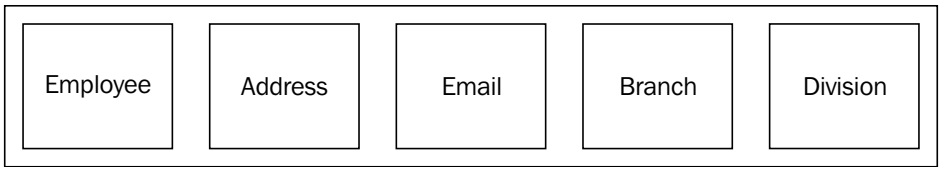
As our data demands increase, we're presented with another problem. If we are storing order information, for example, strictly under the flat file paradigm, we are forced to store a new record each time an order is placed. Consider this example, in which our customer purchases six different items. We store a six-digit invoice number, customer name, and address for the customer's purchase, as follows:

```
487345, Susan, Bates, 123 State St, Somewhere, VA
584793, Susan, Bates, 123 State St, Somewhere, VA
998347, Susan, Bates, 123 State St, Somewhere, VA
126543, Susan, Bates, 123 State St, Somewhere, VA
487392, Susan, Bates, 123 State St, Somewhere, VA
```

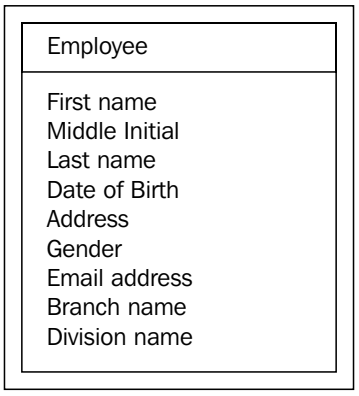
Using this example, notice how much duplicate data we have stored. The fields are invoice number, first name, last name, street address, city, and state, respectively. The only different piece of information in each record is the invoice number, and yet we have repeatedly stored the last five fields—information that is stored in previous records. We refer to these anomalies as repeating values. Repeating values present two problems from a processing standpoint. First, the duplicate data must be re-read each time by our retrieval programs, creating a performance problem for our retrieval operations. Second, those duplicate characters constitute bytes that must be stored on disk, uselessly increasing our storage requirements. It is clear that the flat file paradigm needs to be revised in order to meet the growing demands of our database.

Normalization

The world of databases changed in the early 1970s due in large part to the work of *Dr. Edgar "Ted" Codd*. In his paper, *A Relational Model of Data for Large Shared Data Banks*, Dr. Codd presented a new paradigm—*the relational paradigm*. The relational paradigm seeks to resolve the issues of repeating values and unconstrained size by implementing a process called normalization. During normalization, we organize our data in such a way that the data and its inter-relationships can be clearly identified. When we design a database, we begin by asking two questions—what data do I have? And, how do the pieces of data relate to each other? In the first step, the data is identified and organized into entities. An entity is any person, place, or thing. An entity also has attributes, or characteristics, that pertain to it. Some example entities are listed in the following diagram:



These entities represent distinct pieces of information: the `Employee` entity represents information about employees, the `Email` entity represents information about e-mail addresses, and so on. These entities, and any others we choose to add, make up our data model. We can also look a little closer at the attributes of a particular entity, as shown in the following diagram:



In our example, data (such as `First name`, `Last name`, `Address`, and `Branch name`) are the attributes of the `Employee` entity—they describe information about the employee. This is by no means exhaustive. There would most likely be many more attributes for an employee entity. In fact, this is part of the problem that we discussed earlier with the flat file database—data tends to accumulate, making our file wider and wider, if you will. Additionally, if we were to actually collect this data in a flat file, it might look something like the following screen:

First Name	Mid Initial	Last Name	Address	Email Address
James	R	Johnson	123 State St, Bell, WA	jj@hotmail.com, jj@yahoo.com
Mary	S	Williams	234 First St,Bigtown, VA	mw@gmail.com
Linda	L	Anderson	345 Fifth Ave, Smalltown, MA	la@yahoo.com, la@gmail.com

At first glance, this structure may appear to be adequate, but, if we examine further, we can identify problems with it. To begin with, we note that there are multiple values stored in the `Address` and `Email Address` fields, which can make structuring queries difficult. This is where the process of normalization can assist us. Normalization involves breaking data into different normal forms. These forms are the steps we take to transform non-relational data into relational data. The **first normal form** (**1NF**) involves determining a **primary key**—a value in each occurrence of the data that uniquely identifies it. In the previous example of data, what attribute could be used to uniquely identify each occurrence of data? Perhaps we could use `First Name`.

However, it seems fairly clear that there could be more than one employee with the name James or Mary, so that will not suffice. If we were to use `First Name` and `Last Name` together as our primary key values, we would get closer to uniqueness, but it would still be insufficient for common names such as John Smith. For now, let us say that `First Name`, `Mid Initial`, and `Last Name`, together (as indicated earlier), uniquely identify each occurrence of data and thus comprise our primary key for the employee entity.

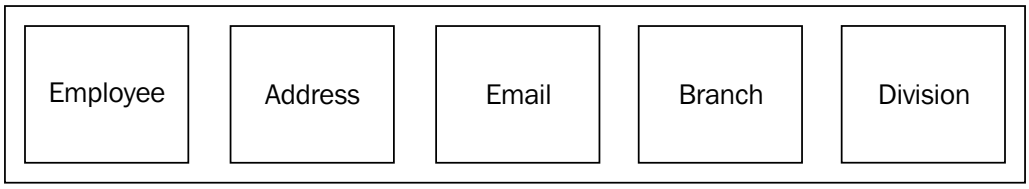
The next issue is the problem of repeating groups. Examine the `Email Address` attribute. It may be required that one or more e-mail addresses be stored for each employee. This presents problems when attempting to query for a particular employee's e-mail address. As each employee can have more than one e-mail address, the `Email` attribute would have to be scanned rather than simply pattern-matched in order to retrieve a particular piece of data. One way to rectify this would be to break each individual occurrence of the e-mail address into two separate records that demonstrates the removal of repeating groups, as shown in the next example. Thus, James R. Johnson, who has two `Email Addresses`, now has two rows in the database—one for the first e-mail address and one for the second:

First Name	Mid Initial	Last Name	Address	Email Address
James	R	Johnson	123 State St, Bell, WA	jj@hotmail.com
James	R	Johnson	123 State St, Bell, WA	jj@yahoo.com
Mary	S	Williams	234 First St,Bigtown, VA	mw@gmail.com
Linda	L	Anderson	345 Fifth Ave, Smalltown, MA	la@yahoo.com
Linda	L	Anderson	345 Fifth Ave, Smalltown, MA	la@gmail.com

We have eliminated the repeating groups, but we have now introduced other problems. First, we have violated our primary key, as first, middle, and last name no longer uniquely identify each row. Second, we have begun to duplicate our data. First name, middle initial, last name, and address are all repeated simply for the sake of removing repeating groups. Lastly, we now realize that it is possible for our employees to have more than one address, which further complicates the problem. Clearly, the first normal form alone is insufficient. It is necessary to transform the data again—this time into the **second normal form** (**2NF**).

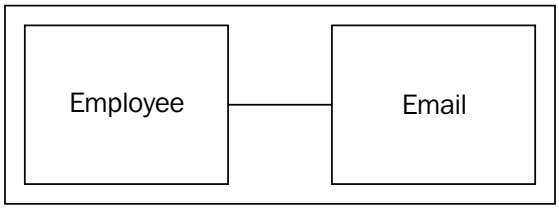
The relational approach

The second normal form involves breaking our employee entity into a number of separate entities, each of which can have a unique primary key and no repeating groups. This is displayed again in the following diagram:

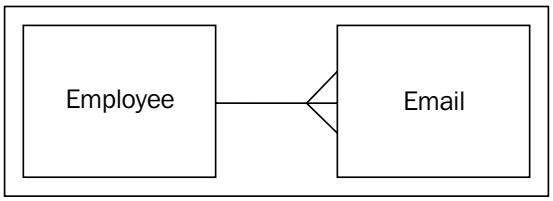


Here, we have separated our employee information into separate entities. We've also added entities that represent the branch and division of which each employee is a part. Now our employee entity contains information such as first name, middle initial, and last name, while our `Email` entity contains the e-mail address information. The other entities operate similarly—each contains information unique to itself.

This may have solved our repeating data problem, but now we simply have five files that have no relation to each other. How do we connect a particular employee to a particular e-mail address? We do this by establishing relationships between the entities; another requirement of the second normal form. A relationship between two entities is formed when they share some common piece of information. How this relationship functions is determined by the business rules that govern our data. Let's say in our model that one, and only one, e-mail address is kept for each employee. We would then say that there is a one-to-one relationship between our `employee` entity and our `Email` entity. Generally, such a relationship is denoted visually with a single bar between the two. We could diagram it as follows:

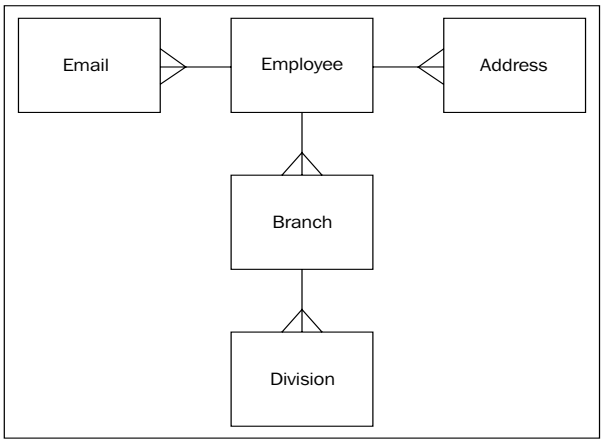


A more realistic relationship, however, would be one where each employee could have more than one e-mail address. This type of relationship is termed a one-to-many relationship. These relationships form the majority of the relationships used in the relational model and are shown in the following diagram. Note the crow's foot connecting to the `Email` entity, indicating many:



As you might expect, there is another type of relationship, one which, under relational rules, we try to avoid. A many-to-many relationship occurs when multiple occurrences of the data in one entity relate to multiple occurrences in the other. For instance, if we allowed employees to have multiple e-mail addresses, but also allowed multiple employees to share a single e-mail address. In the relational paradigm, we seek to avoid these types of relationships, usually by relating an entity between the two that transforms a many-to-many relationship into two distinct one-to-many relationships. The last step in normalization is generally the transformation into the **third normal form** (**3NF**). In the 3NF, we remove any columns that are not dependent on the primary key. These are known as **transitive dependencies**. To resolve these dependencies, we move the non-dependent columns into another table. There are higher normal forms, such as **fourth normal form** (**4NF**) and **fifth normal form** (**5NF**), but these forms are less commonly used. Generally, once we have taken our data structure up to the 3NF, our data is considered relational.

When we design the number and types of relationships between our entities, we construct a data model. This data model is the guide for the DBA on how to construct our database objects. The visual representation of a data model is commonly referred to as an entity relationship diagram (ERD). Using the five example entities we listed previously, we can construct a simple entity relationship diagram, as demonstrated in the following example:



From this example model, we can determine that the `employee` entity is more or less the center of this model. An employee can have one or more e-mail addresses. An employee can also have one or more street addresses. An employee can belong to one or more branches, and a branch can belong to one or more divisions. Even though it is highly simplified, this diagram shows the basic concepts of visually modeling data.

Through the use of relational principles and entity relationship diagrams, a database administrator or data architect can take a list of customer data and organize it into distinct sets of information that relate to one another. As a result, data duplication is greatly reduced and retrieval performance is increased. It is because of this efficient use of storage and processing power that the RDBMS is the predominant method used in storing data today.

SQL in the real world

Strictly speaking, the Oracle RDBMS is actually an **Object Relational Database Management System** (**ORDBMS**) and has been since Oracle version 8. An ORDBMS refers to the ability of Oracle databases to be manipulated using object-oriented concepts.

Bringing it into the Oracle world

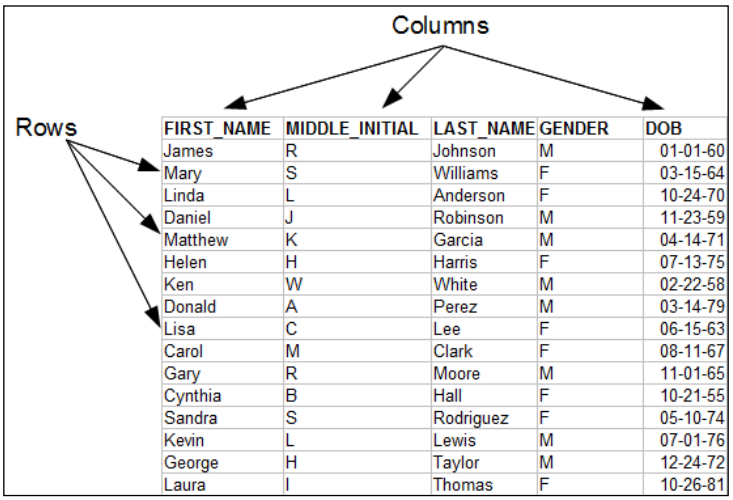
To discuss the relational paradigm, we have used relational terminology, which is designed to be generic and not associated with any particular database product. The subject of this book, however, is using SQL with Oracle databases. It is time to relate the terminology used in the relational paradigm to terms that are likely more familiar:

Relational	**Flat file**	**Oracle-specific**
Entity	File	Table
Attribute	Field	Column
Tuple	Record	Row

The preceding diagram shows a comparison table of the different terms used to describe basic database components. Up to this point, we have used the relational term, entity, to describe our person, place, or thing. From this point, we will refer to it by its more commonly known name—the table.

Tables and their structure

If you've ever used a spreadsheet before, then you are familiar with the concept of a table. A table is the primary logical data structure in an Oracle database. We use the term logical because a table has no physical structure in itself—you cannot simply login to a database server, open up a file manager, and find the table within the directories on the server. A table exists as a layer of abstraction from the physical data that allows a user to interface with it in a more natural way. Examine the following diagram; like a spreadsheet, a table consists of columns and rows:



FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	GENDER	DOB
James	R	Johnson	M	01-01-60
Mary	S	Williams	F	03-15-64
Linda	L	Anderson	F	10-24-70
Daniel	J	Robinson	M	11-23-59
Matthew	K	Garcia	M	04-14-71
Helen	H	Harris	F	07-13-75
Ken	W	White	M	02-22-58
Donald	A	Perez	M	03-14-79
Lisa	C	Lee	F	06-15-63
Carol	M	Clark	F	08-11-67
Gary	R	Moore	M	11-01-65
Cynthia	B	Hall	F	10-21-55
Sandra	S	Rodriguez	F	05-10-74
Kevin	L	Lewis	M	07-01-76
George	H	Taylor	M	12-24-72
Laura	I	Thomas	F	10-26-81

A column identifies any single characteristic of a particular table, such as first name. A column differs from a row in more than its vertical orientation. Each value within a column contains a particular type of data, or data type. For instance, in the preceding example, the column `FIRST_NAME` denotes that all data within that column will be of the same type and that data type will be consistent with the label `FIRST_NAME`. Such a column would contain only character string data. For instance, in the `FIRST_NAME` column, we have data such as `Mary` and `Matthew`, but not the number 42532.84. In the date of birth column, or `DOB`, only date data would be stored. As we will see in the next chapter, in Oracle, string data or text data is not the same thing as date data.

Along the horizontal, we have rows of data. A row of data is any single instance of a particular piece of information. For example, in the first row of the table in our example, we have the following pieces of information:

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

```
First name = "James"
Middle initial = "R"
Last name = "Johnson"
Gender = "M"
DOB = "01-01-60"
```

This information comprises the sum total of all the information we have in this table for a single individual, namely, James R. Johnson. The following row, for Mary S. Williams, contains the same types of information, but different values. This construct allows us to store and display data that is orderly in terms of data types, but still flexible enough to store the data for many different individuals. Together, the columns and rows of data form a relational table: the heart of the Oracle database. However, in order to retrieve and manipulate this table data, we need a programming language; for relational databases, that language is SQL.

Structured Query Language

SQL was developed by Donald Chamberlain and Raymond Boyce in the early 1970s as a language to retrieve data from IBM's early relational database management systems. It was accepted as a standard by the **American National Standards Institute** (**ANSI**) in 1986. SQL is generally referred to as a **fourth-generation language** (**4GL**), in contrast with **third-generation languages** (**3GLs**) such as C, Java, and Python. As a 4GL, the syntax of SQL is designed to be even closer to human language than 3GLs, making it relatively natural to learn. Some do not refer to SQL as a programming language at all, but rather a data sub-language.

A language for relational databases

Before we look at what SQL (pronounced either 'S-Q-L' or 'sequel') is, it is important to define what it is not. First, SQL is not a product of Oracle or any other software company. While most relational database products use some implementation of SQL, none of them own it. The structure and syntax of SQL are governed by the American National Standards Institute and the **International Organization for Standardization** (**ISO**). It is these organizations that decide, albeit with input from other companies such as Oracle, what comprises the accepted standard for SQL. The current revision is SQL:2008.

Second, while the ANSI standard forms the basis for the various implementations of SQL used in different database management systems, this does not mean that the SQL syntax and functionality in all database products is the same; in fact, it is often quite different. For instance, the SQL language permits the concatenation of two column values into one; for example, the values *hello* and *there* concatenated would be *hellothere*. Oracle and Microsoft SQL Server both use symbols to denote concatenation, but they are different symbols. Oracle uses the double-pipe symbol, '||', and SQL Server uses a plus sign, '+'. MySQL, on the other hand, uses a keyword, `CONCAT`. Additionally, RDBMS software manufacturers often add functionalities to their own SQL implementations. In Oracle version 10g, a new type of syntax was included to join data from two or more tables that differs significantly from the ANSI standard. Oracle does, however, still support the ANSI standard as well.

SQL in the real world

Although the SQL implementations of the major RDBMS products differ, they all conform to the basic ANSI standard. That means if you learn how SQL is used in one database product, such as Oracle, much of your acquired knowledge should transfer easily to other database products.

Last, SQL should not be confused with any particular database product, such as Microsoft SQL Server or MySQL. Microsoft SQL Server is sometimes referred to by some as SQL; a confusing distinction.

What SQL does provide for developers and database administrators is a simple but rich set of commands that can be used to do the following:

- Retrieve data
- Insert, modify, and delete data
- Create, modify, and delete database objects
- Give or remove privileges to a user
- Control transactions

One of the interesting things about SQL is its dataset-oriented nature. When programmers use third-generation languages such as C++, working with the kinds of datasets we use in SQL is often a cumbersome task involving the explicit construction of variable arrays for memory management. One of the benefits of SQL is that it is already designed to work with arrays of data, so the memory management portion occurs implicitly. It is worth noting, however, that third-generation languages can do many things that SQL cannot. For instance, by itself, SQL cannot be used to create standalone programs such as video games and cell phone applications. However, SQL is an extremely effective tool when used for the purpose for which it was designed—namely, the retrieval and manipulation of relational data.

SQL in the real world

Standard programming constructs such as flow control, recursion, and conditional statements are absent from SQL. However, Oracle has created PL/SQL, a third-generation overlay language that adds these and other basic programmatic constructs to the SQL language. Because of the strength of the SQL language in manipulating data, PL/SQL is often the choice of developers when programming the portions of their applications that interact with Oracle databases.

The goal of this book is to teach you the syntax and techniques to use the SQL language to make data do whatever you want it to do. *Chapter 2, SQL SELECT Statements* and beyond address these topics. However, before we can learn more about the SQL language, we are going to need to choose a tool that can interact with the database and process our SQL.

Commonly-used SQL tools

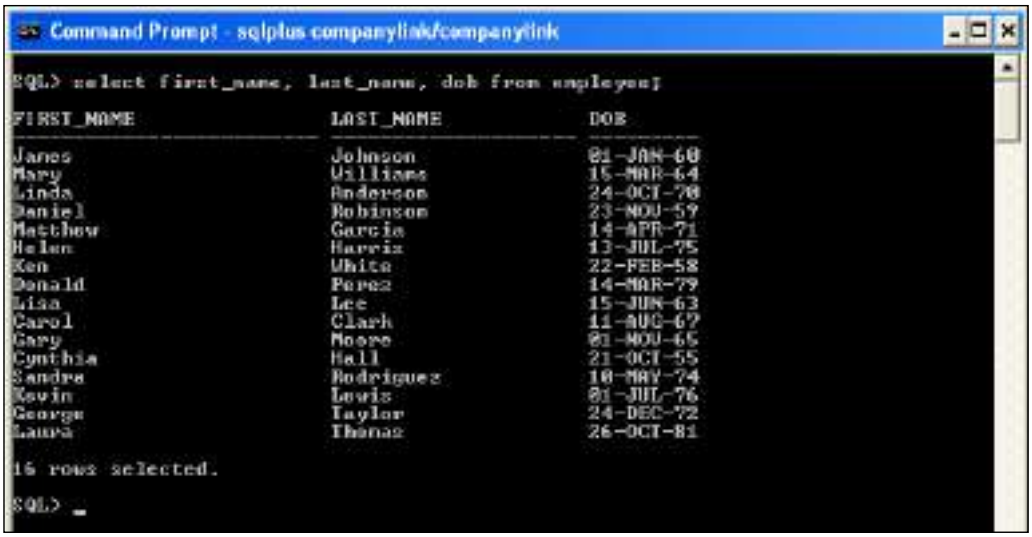
Because SQL is the primary interface into relational databases, there are many SQL manipulation tools from which to choose. There are benefits and drawbacks to each, but the choice of tool to use is generally about your comfort level with the tool and its feature set. Some tools are free, some are open source, and some require paid licenses; however, each tool uses the same syntax for SQL when it connects to an Oracle database. Following are some commonly-used SQL tools:

SQL in the real world

While your choice of SQL tool is an important one, in the industry it is one that is sometimes dictated by the toolset standards of your employer. It's important that you don't completely dedicate yourself to one tool. If you become an expert at one and then transfer to a different employer whose standards don't allow for the use of your tool, you may find yourself with an initial learning curve.

SQL*Plus

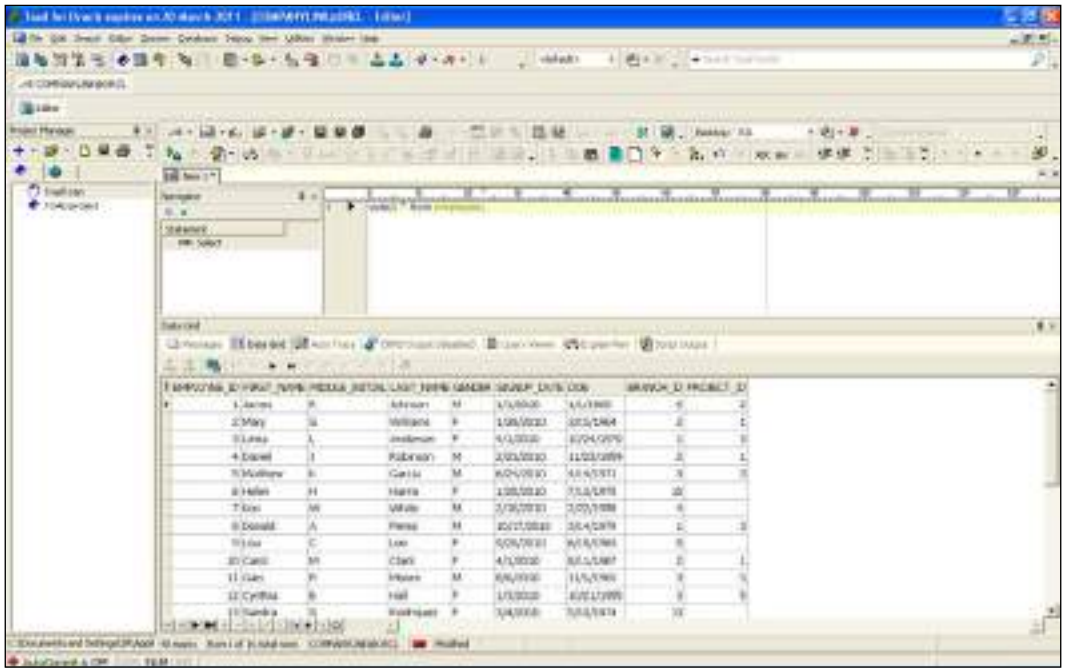
SQL*Plus is the de facto standard of SQL tools to connect to an Oracle database. Since Oracle's early versions, it has been included as a part of any Oracle RDBMS installation. SQL*Plus is a command-line tool and is launched on all Oracle platforms using the command, `sqlplus`. This command-line tool has been a staple of Oracle database administrators for many years. It has a powerful, interactive command interface that can be used to issue SQL statements, create database objects, launch scripts, and startup databases. However, compared with some of the newer tools, it has a significant learning curve. Its use of line numbering and mandatory semicolons for execution is often confusing to beginners. Oracle has also released a GUI version of SQL*Plus for use on Windows systems. Its rules for use, however, are still generally the same as the command-line interface, and its confinement to the Windows platform limits its use. As of Oracle version 11g, the GUI version of SQL*Plus is no longer included with a standard Oracle on Windows installation. Whatever your choice of SQL tool, it is very difficult for a database administrator to completely avoid using SQL*Plus. The following is a screenshot of the command-line SQL*Plus tool:



TOAD

The **Tool for Oracle Application Developers** (**TOAD**) is a full-featured development and administration tool for Oracle as well as other relational database systems, including Microsoft SQL Server, Sybase, and IBM's DB2. Originally created by Jim McDaniel for his own use, he later released it as freeware for the Oracle community at large. Eventually, Quest Software acquired the rights for TOAD and began distributing a licensed version, while greatly expanding on the original functionality. TOAD is immensely popular among both DBAs and developers due to its large feature set. For DBAs, it is a complete administration tool, allowing the user to control every major aspect of the database, including storage manipulation, object creation, and security control. For developers, TOAD offers a robust coding interface, including advanced debugging facilities.

TOAD is available for download in both freeware and trial licensed versions. A screenshot is shown as follows:

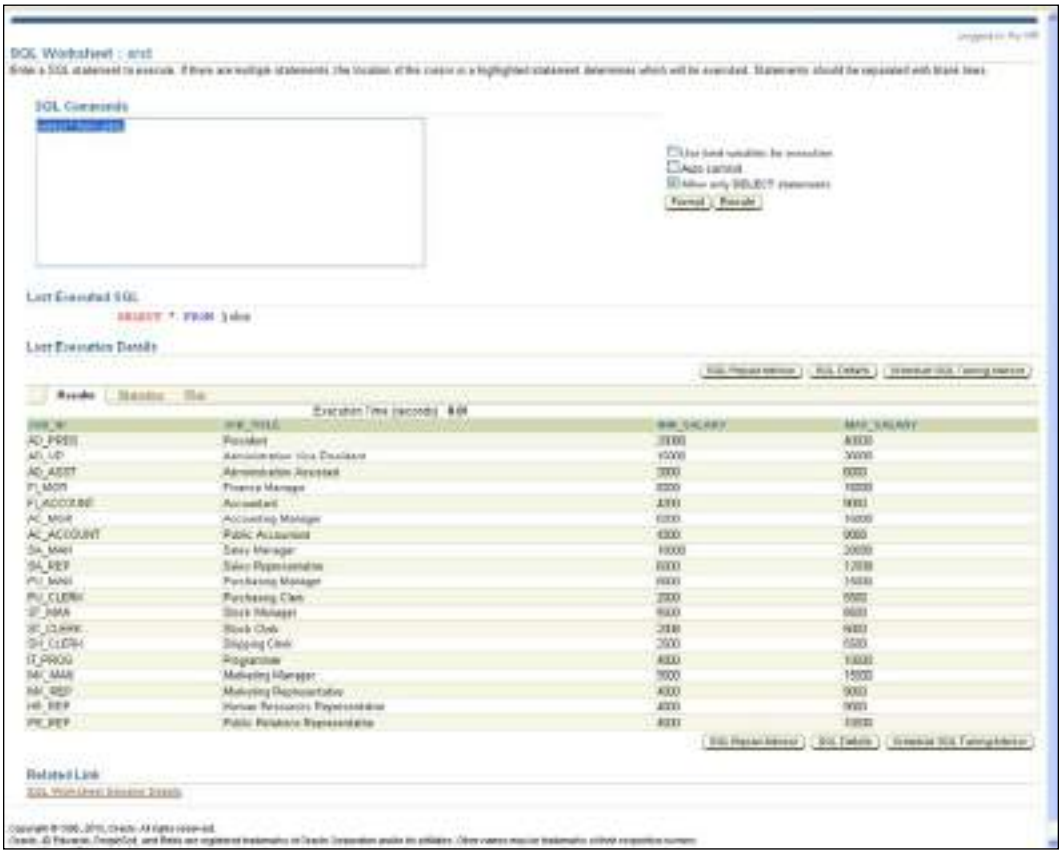


DBArtisan

DBArtisan (now called DBArtisan XE), by Embarcadero Technologies, is another complete suite of database management tools that operates across multiple platforms. DBArtisan is only available as a licensed product, but has extensive administration capabilities, including the ability to do advanced capacity and performance management, all packaged in an attractive and user-friendly GUI frontend. A trial version is available for download from Embarcadero's website.

SQL Worksheet (Enterprise Manager)

The **SQL Worksheet** is not a separate tool in itself; rather, it is a component of the larger Enterprise Manager product. Enterprise Manager is Oracle's flagship, web-based administration suite, comprised of two main components—Database Control and Grid Control. Database Control operates from a single server as a Java process and allows the DBA to manage every aspect of a single database, including storage, object manipulation, security, and performance. Grid Control operates with the same GUI interface, but requires the installation of the Enterprise Manager product on a centralized server. From this central instance of Grid Control, a DBA can manage all the databases to which Grid Control connects, providing the DBA with a web-based interface to the entire environment. **SQL Worksheet**, a link within Database/ Grid Control, provides a basic SQL interface to a database.. The license for both Grid Control and Database Control is included in the license for the Enterprise edition of Oracle, although many of the performance tuning and configuration management features must be separately purchased. A screenshot of SQL Worksheet is shown as follows:



PL/SQL Developer

PL/SQL Developer is a full-featured SQL development tool from Allround Automations. Along with many of the other features common to SQL development tools, such as saved connections, data exporting, and table comparisons, PL/SQL Developer places a strong focus on the coding environment. It offers an extensive code editor with an integrated debugger, syntax highlighting, and a code hierarchy that is especially beneficial when working with the PL/SQL language. It also includes a **code beautifier** that formats your code using user-defined rules. PL/SQL Developer can be purchased from Allround Automations or downloaded from their website as a fully-functional, 30-day trial version.

Oracle SQL Developer

Oracle SQL Developer, originally called **Raptor**, is a GUI database interface that takes a somewhat different approach from its competitors. While many of the major licensable GUI administration products have continued to expand their product offering through more and more add-on components, SQL Developer is a much more dedicated tool. It's a streamlined SQL interface to the Oracle database. You can create and manipulate database objects in the GUI interface as well as write and execute SQL statements from a command line. Administration-oriented activities such as storage control are left to Enterprise Manager. SQL Developer aims to be a strong SQL and PL/SQL editor with some GUI functionalities. SQL Developer has gained in popularity in recent years, in large part to several benefits that are listed as follows:

- It is completely free with no mandatory licensable components, although third-party add-ons are available for purchase.
- It is a true cross-platform client-side tool written primarily in Java. While a majority of the commonly-used SQL tools are available only on the Windows platform, SQL Developer runs on Windows, Linux, and even the Mac.
- In many Oracle shops, DBAs have been uncomfortable with the idea of giving developers a tool that can be used to cause massive damage to the database. Because Oracle has separated out most of the administration functions from SQL Developer, it is more of a true development tool.
- SQL Developer supports read-only connections to many popular databases, including SQL Server, Sybase, MySQL, Microsoft Access, DB2, and Teradata.
- Because it is written in Java, it allows for the creation and addition of third-party extensions. If you want a capability that SQL Developer does not have, you can write your own!

- It is provided by Oracle and is now included with any installation of Oracle database. It has essentially replaced SQL*Plus as Oracle's default SQL interface, although SQL*Plus is still available from the command line.

For these reasons, the tool we use in this book for the purposes of demonstration will be SQL Developer. Instructions for downloading the tool are in the foreword. But, before we look at SQL Developer, let's find out a little about the data we'll be using and look at the `Companylink` database.

Working with SQL

'Often, the best way to learn something is hands-on. To best facilitate this, we will use a scaled-down set of data that mirrors the type of data used in the real world.

Introducing the Companylink database

This book focuses on two objectives:

1. To prepare you for the 11g SQL Fundamentals I exam (Oracle exam #1Z0-051).
2. To present the knowledge needed for the exam in such a way that you can use it in a real-world setting.

To that end, rather than using the default tables included in Oracle, we will be working with simulated real-world data. The database we will use throughout this book is for the fictional company, Companylink. Although most people are aware of the impact of social networking in our private lives, companies are realizing the importance of using it in their industries as well. Companylink is a business that focuses on social networking in the corporate setting. The data model that we will use is a small but realistic set of working data that could support a social networking website. The following tables are included in the `Companylink` database, which can be downloaded from Packt support site as well as comments about the business rules that constrain them:

- **Employee**: Information about employees that use the Companylink site.
- **Address**: The street address information.
- **Branch**: The corporate branch to which each employee belongs. Each employee belongs to one branch.
- **Division**: It is the corporate division to which each branch belongs. Each division is associated with multiple branches.
- **E-mail:** An employee can store multiple e-mail addresses.

- **Message**: Our fictional Companylink social networking site allows you to send messages to fellow employees. That information is stored here.
- **Website**: Companylink allows users to create their own personal web pages. The URL of these pages is contained in this table.
- **Blog**: In addition to a website, users can optionally create their own blogs. This information is stored in the `Blog` table.
- **Project**: Each employee is assigned to a single primary project, which is contained here.
- **Award**: Employees can win corporate awards. The list of possible awards is stored here. Employees can win more than one award.
- **Employee_award**: This table is used to relate employees with their awards. Since multiple employees can win the same award and multiple awards can be won by the same employee, this creates a many-to-many table relationship, which, in the relational paradigm, must be avoided. The `employee_award` table divides this many-to-many relationship into two distinct one-to-many relationships.

To create our database, we need to run the downloaded Windows command file. Simply unzip the `companylink.zip` file into a directory and double-click on the `companylink_db.cmd` file. The execution of the file will do the following:

- Make a connection to the database
- Create a user called companylink with the password companylink
- Create the tables used for the examples in this book
- Populate these tables with data
- Output two log files: `companylink_ddl.txt` and `companylink_data.txt`

If you wish, the log files can be used to verify successful execution of the script. The command file is completely reusable, which is to say that if you break any of the tables or data, you only need to disconnect from the database and double-click the command file again. It will drop the existing data and rebuild the tables from scratch. When you do this, keep in mind that any data you add yourself will be deleted as well. Throughout the book, we will continually be writing SQL statements that access these tables and will even add new ones.

The creation of these tables requires a working installation of the Oracle database software on a machine to which you have access. Fortunately, the Oracle software can be downloaded from `http://www.oracle.com/technology`. There is no purchased license required if you use the software for your own learning purposes.

SQL in the real world

When you're starting out with SQL and Oracle, it's important to get hands-on. Although Oracle makes its software available at no charge for personal use, many aspiring DBAs are hesitant to install it on their personal computers. By using free desktop virtualization software, such as Virtualbox, you can create a virtual machine on your home computer that can be used as your self-contained database server. Whenever you want to work with Oracle, simply start your virtual machine. Whenever you finish, shutdown the virtual machine, and all the resources it used will be released. Virtualization can be a useful solution to isolate your Oracle work from your home use without buying another computer.

An introduction to Oracle SQL Developer

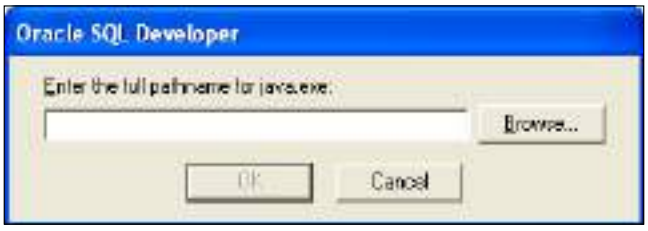
Since SQL Developer is our SQL tool of choice, its important that we get a good feel for it right from the beginning. In this section, we learn about configuring and running SQL Developer.

Setting up SQL Developer

Let's get started with SQL Developer. If you have Oracle installed, you can launch SQL Developer in Windows XP from the **Start** menu, as shown next:

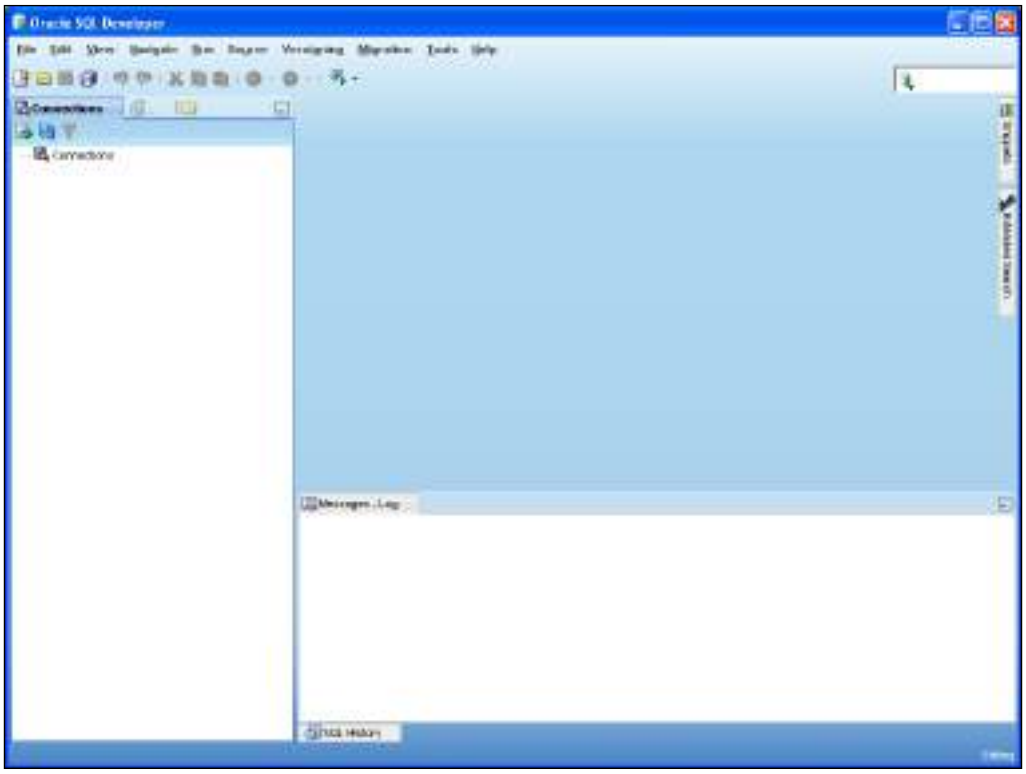
Start | All Programs | Oracle<program group> | Application Development | SQL Developer

SQL Developer runs as a Java application, so it may take a little while to load. The first time you start the application, you may get a message box, such as the one shown in the following screenshot:



If this happens, click on **Browse** and navigate to the `java.exe` file. You do not need a separate installation of Java to run SQL Developer; one is included in the Oracle installation. If you don't know where the `java.exe` is located, simply go to the Oracle installation directory and do a search for `java.exe`. Then, navigate to that path and select it.

Once startup has completed, you will see a **Tip of the Day screen**. Close it and you will be presented with the following screen. It's worth noting that this screen will look the same, irrespective of whether you're running SQL Developer under Windows, Linux, or the Mac OS, due to its cross-platform, Java-based nature.



On the left side, you see a list of connections to databases. At this point, there will be no connections, since we have not created any yet. SQL Developer allows you to maintain multiple connections to various databases. Each one can use any variation of different login names, servers, or database names.

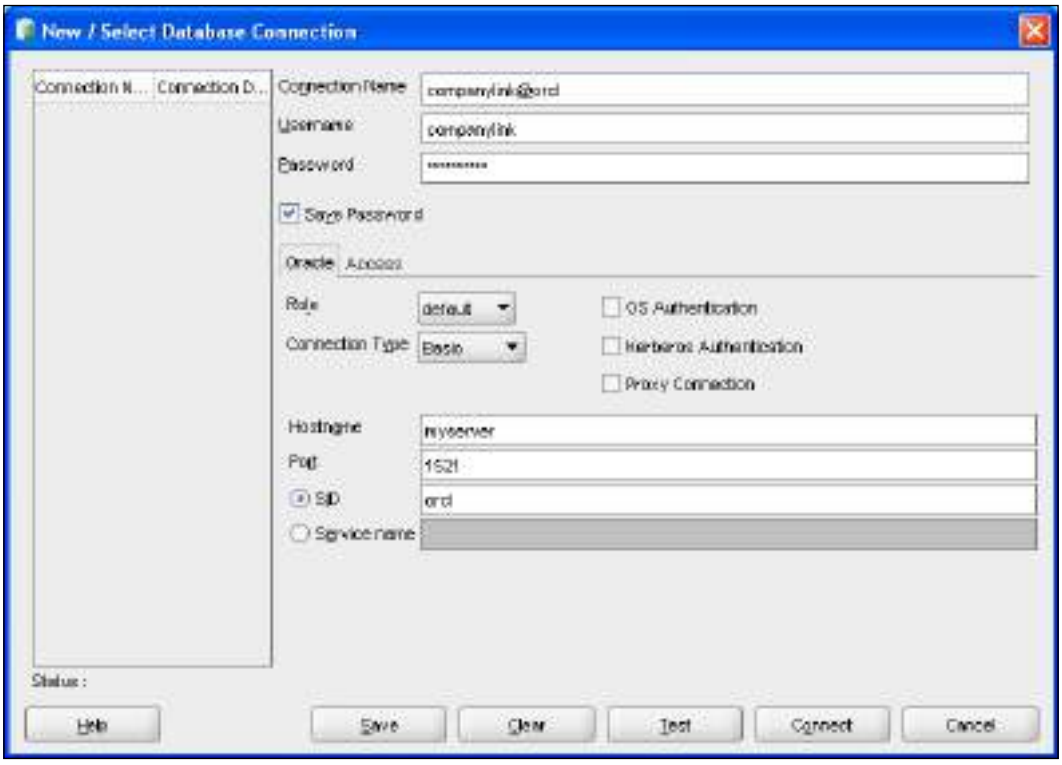
SQL in the real world

In the real world, DBAs and developers run SQL Developer from their desktops and use it to connect to remote databases. Thus, their working environments can run locally, but the databases they connect to can be anywhere in the world!

Before we can use SQL, we need to connect to a database. To do that, we need to create a database connection. Any connection to an Oracle database consists of three pieces of information:

1. The hostname or IP address of the machine to which we're connecting.
2. The port number on which Oracle operates.
3. The name of the database to which we connect.

To set up our connection, we need to click on the **New Connection** button at the top of the left-hand connection frame. This action brings up the **New / Select Database Connection** window. We fill in the information, as listed in the following screenshot. This example connection assumes that you have set up an Oracle database using the standard installation procedure with common defaults. If you're connecting to an existing database, the information you enter will be different:



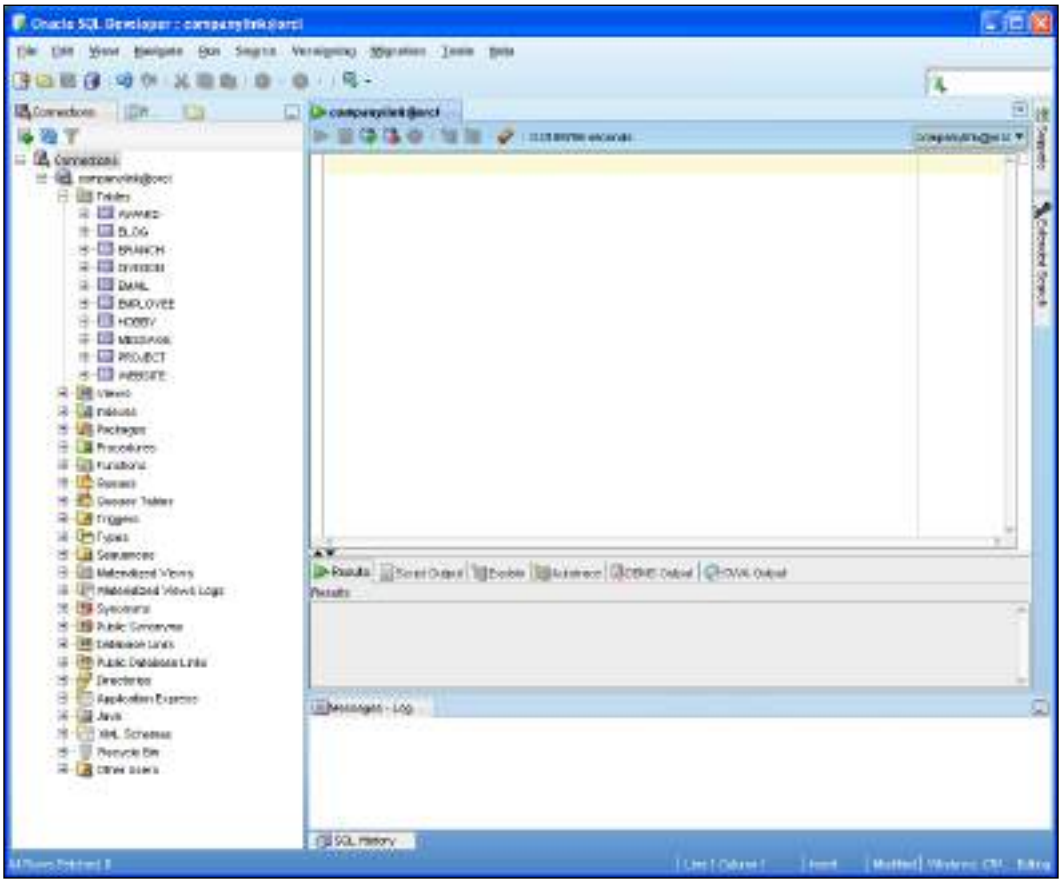
The pieces of information that are relevant to us are as follows:

- **Connection name**: This can be whatever we choose, but it is usually a good idea to make it descriptive of the connection itself. In our example, we choose `companylink@orcl` because it denotes that we are connecting to the `orcl` database as the `companylink` user.
- **Username**: The name of the user we connect as.
- **Password**: The password for the user. The password for our user is companylink (non-case sensitive)
- **Save Password**: Select this checkbox to ensure that you don't have to re-enter the password each time you initiate the connection.
- **Hostname**: This will be either the hostname or the IP address of the server that hosts our target database. The example used, `myserver`, will most likely not be the name of your server. Change this to the name relevant to your situation.
- **Port**: This will be the port number that Oracle is running from. Most Oracle databases run from port 1521, although some DBAs change this for security reasons. If you installed Oracle using the default settings, your port number will be 1521.
- **SID**: The SID is the System Identifier for your database, which is the name of the database. In a typical installation of Oracle, the default SID used is `orcl`.

Once the relevant information has been entered, it is always a good idea to click on the **Test** button at the bottom of the window to ensure a connection can be made. If all the information is correct, you should see **Status: Success** on the lower left-hand side of the window. Once we have verified that we can successfully connect, we click on the **Connect** button. Our connection is saved in the **Connections** frame, on the left side of the window, and our connection is established.

Getting around in SQL Developer

Now that we're connected, let's take a look at what SQL Developer has to offer. Click on the plus sign (**+**) next to your new connection:

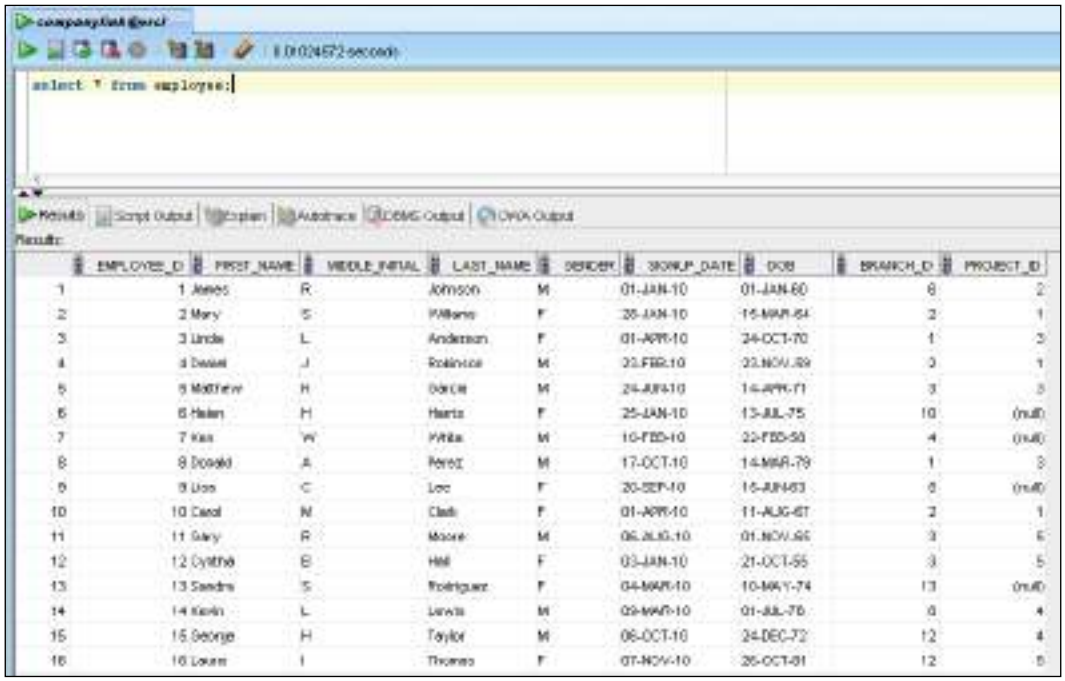


On the left side, indented under our connection, is a list of database objects, including **Tables**, **Views**, and **Indexes**. Discussion about some of the other sets of objects is outside the scope of this book, but all are accessible by simply expanding the object group using the plus sign next to it. Click on the **+** next to **Tables** and your list of tables will be expanded. Your window should now look something similar to what is shown here. These are the tables created, and therefore owned, by the user with whose profile we logged in; in this case, `companylink`. They were created by running `companylink.bat`, earlier in the chapter. The following screenshot shows a list of our `Companylink` tables:

These tables can be expanded to view their characteristics, such as column names and datatypes, but most of this book will focus on how to view and modify tables using only the SQL language instead of GUI tools.

The large portion of the window in the upper right is our SQL working area. This frame will be the area in which we write SQL code. To write SQL in the working area, simply click in the area and begin typing your SQL statements. When you are finished, click on the green arrow in the working area toolbar to execute the statement. Alternatively, you can press *F9* on your keyboard.

Directly below the working area is the **Results** frame. This is the area where we will see the results of our SQL queries. The results will display in columnar format, and the columns can be resized by clicking-and-dragging. The **Results** frame also has several tabs across the top for various other functions, but, for now, we will not concern ourselves with them. Let's try a query and view the output. In the working area, type the SQL query you see in the following screenshot, **select * from employee,** and click on the green execute arrow:



As we will learn in the next chapter, the SQL query we've placed in the working area uses a wildcard character, `'*'`, to display all the columns and rows from the table called `employee`. As you can see, this data displays in the **Results** frame, which is listed in columnar format. You have just made your first use of the Structured Query Language.

Below the **Results** frame is the **Messages (Log)** frame. It is used to display the output of certain operations and is not relevant to our concerns. To maximize the areas for the working area and **Results** frame, you can click-and-hold the bar above the **Messages** frame and drag it downward to make it invisible. Similarly, you can click-and-drag the bar between the working area and **Results** frames to change the ratio of space between the two. Many users like to make as much of the **Results** frame visible as possible so as to see more of the resultant data.

The last area we need to point out is the **SQL History** tab just below the **Messages** frame. This tab, when clicked, displays a pop-up of the most recent SQL statements. This can be very useful when trying to remember previous statements. Simply click on the tab, then double-click the statement you want to run, and it will be pasted in the working area. You can then select it and click on **Execute** to run it.

SQL Developer offers a tremendous number of other features that are beyond the scope of this book. If you're interested in more information on SQL Developer, visit `http://www.oracle.com/technology` and view the documentation for it.

Summary

In this chapter, we've gone from the early days of databases to the relational databases that are so prolific today. We've explored the concept of normalization and how it's applied to the relational paradigm. We've looked at tables and how they are structured and introduced the Structured Query Language for relational databases. We've also examined some of the popular SQL tools and created the tables needed for the `Companylink` database. Finally, we've worked our way around the SQL Developer tool and learned the basics of how to execute queries.

Now that we've learned about relational databases and SQL, we're ready to begin writing SQL statements—the topic of the next chapter.

Test your knowledge

1. What relational term is used to denote any person, place, or thing?
    a. Attribute
    b. Entity
    c. Flat file
    d. Repeating group
2. What is the name of the process used to transform non-relational data into relational data?
    a. Normalization
    b. Transformation
    c. Repudiation
    d. Object-oriented

3. A _____ uniquely identifies any single row of data.
    a. Foreign key
    b. Attribute
    c. Primary key
    d. Column
4. Which of these is NOT a form of entity relationship?
    a. One-to-many
    b. One-to-one
    c. Variant-to-one
    d. Many-to-many
5. What is the visual representation of a data model called?
    a. A table
    b. An entity
    c. Normalization
    d. An entity relationship diagram
6. Which of these is NOT required to make a database connection?
    a. Port number
    b. Table name
    c. Database name
    d. Hostname/IP Address

2
SQL SELECT Statements

In the previous chapter, we laid the groundwork for using SQL with Oracle databases. Now, we are almost ready to begin writing our own SQL statements. But, first, we'll need to examine the rules. Once that's complete, we will proceed to learn several different basic SQL queries, all derived from the single, most important SQL statement at our disposal—the `SELECT` statement. With this statement, we can retrieve data from an Oracle database and, by the end of the chapter, will begin to do transformations of the data as well. Beginning with this chapter, we also make note of the *Certification objectives covered*. This is a guide for us to match our subject matter with the objectives of the certification exam.

In this chapter, we shall:

- Understand the purpose of `SELECT` statements
- Explore the syntax and usage of `SELECT` statements
- Use `SELECT` for single-column data retrieval
- Use `SELECT` for multi-column data retrieval
- Use `SELECT` to retrieve all the columns in a table
- Display table structure using `DESCRIBE`
- Examine aliases and their uses
- Utilize SQL with arithmetic operators to execute mathematical computations
- Understand the concept of NULL values
- Retrieve unique values using `DISTINCT`
- Display concatenated values

The purpose and syntax of SQL

In the beginning of the previous chapter, we discussed the importance of data in our everyday lives and gave examples of the kinds of data stored in databases. We then created our own data for the `Companylink` database and discussed a little about its structure. However, sitting at rest within a database, that data is of little practical use. It is not enough to simply store data; it must be retrieved and manipulated to be useful. For instance, in order to use the data in the `Companylink` database, we need to be able to perform practical operations such as the following:

- Retrieve a list of the employees on *Companylink*, along with their street addresses, for a company mailing list
- Retrieve employee names and their date of birth for a Happy Birthday application
- Add and store new messages that employees send to one another
- Change an employee's primary project when they are transferred to a new one
- Remove the e-mail address of an employee when it is no longer valid
- Add structures that will hold entirely new functionalities for *Companylink*, such as hobby lists or friend finders
- Remove the structures of functionalities that are no longer needed
- Control security to employee data

SQL can be used to do all these things, and more. With this relatively simple, semantically-familiar language, we can retrieve and manipulate data and database objects in ways that make it useful for our company. Using SQL, we can essentially "make the data do what we want". But, first, we need to understand the rules.

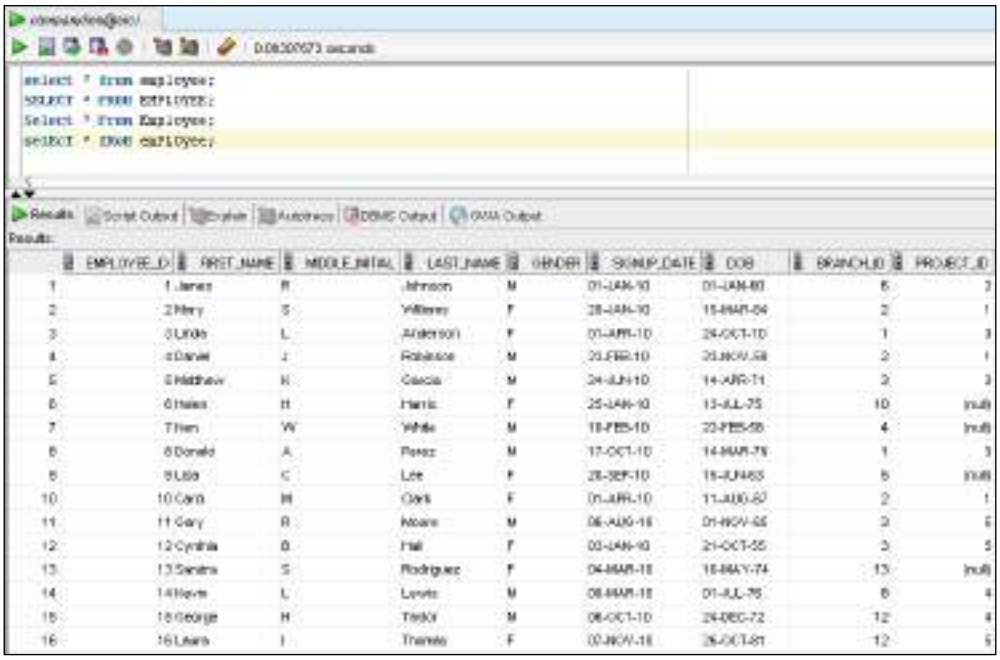
The syntax of SQL

If you have ever written in a programming language before, then think back to the first one you learned. If you're like most, then one of the most frustrating experiences in learning programming languages is how incredibly precise they are. The keywords, symbols, and structure must all be "just right" in order for the program code to execute. This contrasts significantly with our own human languages, which are comparatively imprecise. Our day-to-day language is filled with references, slang, innuendo, and words that have more than one meaning. It serves us well for interpersonal communication, but it would be incredibly inefficient to speak to a computer in that way, since any commands we give to a computer must be translated into machine code in order to be understood. If programming languages were structured like human languages, then that process of translation would

be significantly more complex, since human languages have a certain degree of ambiguity. On the other hand, programming languages must still be comprehensible to humans in order to be useful. These two considerations come together in the syntax of any programming languages. The **syntax** of a programming language is the set of rules that define its structures, symbols, and semantics. SQL entered into any tool must be, first and foremost, syntactically correct; a failure to do so will result in an error.

Case sensitivity

Many programming languages are case-sensitive, denoting that instructions are interpreted differently, depending on whether individual characters are uppercase or lowercase. In Oracle's implementation of SQL, this is not the case. SQL commands have the same meaning whether used with uppercase or lowercase characters. Thus, each of these statements in the following screenshot are equivalent:

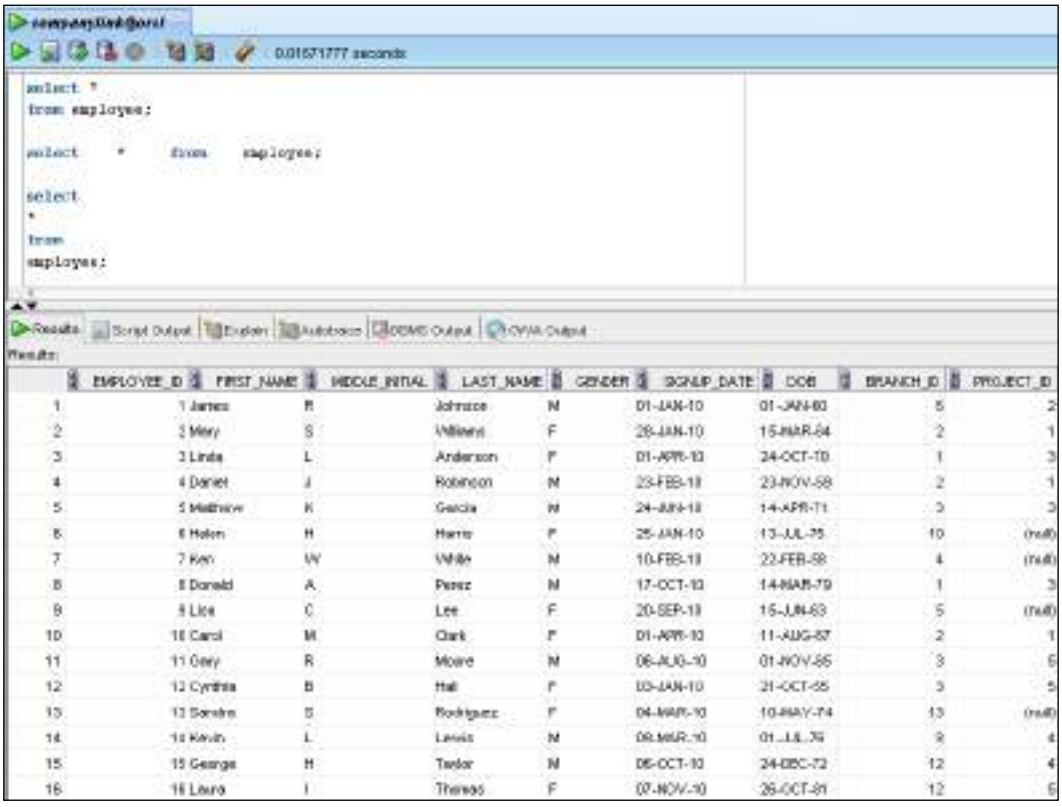


It is important to note that the case-insensitivity of SQL in Oracle is not dependent on which operating system is used. This is true of both the client operating system as well as the operating system of the server that hosts the database. The way in which the operating system treats case has no effect on SQL statements, since they are executed from within an SQL tool such as the ones that we listed in the previous chapter, and not the operating system command line itself.

There is one important exception to this case-insensitivity; namely, with the use of quotes around certain elements of SQL statements. We will investigate this further, later in the chapter.

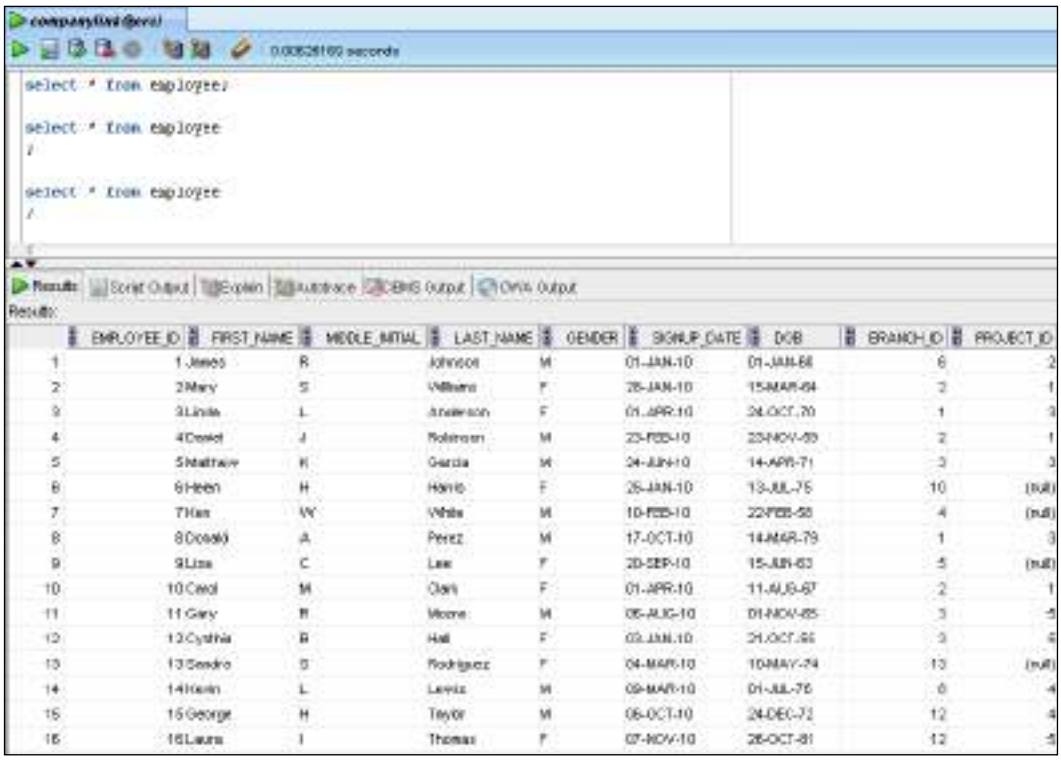
The use of whitespace

The use of whitespace is often strictly governed in the syntax of programming languages. In programming code, **whitespace** is the term used to describe various non-printing characters in a line of code, such as tab characters, spaces, and end-of-line characters. For example, the Python language uses indentation to establish the structure of the command. SQL is much more forgiving in terms of the use of whitespace. Although the various SQL elements, or "words", must be separated by whitespace (usually a "space" character), the use of extra spaces, tabs, and end-of-line characters has little effect on the syntactical correctness of the statement. Thus, in the following screenshot, we see statements that use whitespace differently, yet are all syntactically interpreted as the same:



Statement terminators

Statement terminators are used by programming languages to distinguish the end of a particular statement. This allows the use of multiline statements, such as the one shown in the previous example. In Oracle SQL, two statement terminators can be used; the semicolon (;) and the forward slash (/). The two are similar in their use, the main difference being that the forward slash can only be used on a separate line. Examine the use of the semicolon and forward slash as statement terminators in the following screenshot:



It is important to note that the SQL Developer tool will allow you to execute individual statements without any terminator at all. However, neglecting the use of statement terminators is a very bad habit to develop, since they must be used in any multi-command scripting that you do. In addition, command-line tools, such as SQL*Plus, require the use of statement terminators, so it's best to establish a habit of using them.

SQL in the real world

In software development, if you're manipulating data, then it is much more common to use a script, a text file that contains a set of SQL commands, than it is to execute individual statements. If you're issuing database manipulation commands as a **DBA** (**Database Administrator**), then it's common to use both. You can think of an SQL script as a way to execute SQL commands in a "batch".

Retrieving data with SELECT statements

One of the most important operations done while connected to a database is the query. In database terms, a **query** is a request to retrieve data from a database table or tables. Queries can range in complexity from simple queries, consisting of a few lines, to extremely long and intricate reports composed of pages of code. Queries form the backbone of business reporting and, as such, are a vital part of the SQL language. We compose queries using the `SELECT` statement, as discussed in this section.

Projecting columns in a SELECT statement

Now that we have laid out some of the syntactical rules for SQL statements, it is time to learn about the structure of our first SQL statement. We see what is commonly referred to as a "syntax tree" for a `SELECT` statement, shown as follows:

```
SELECT {column, …} [*] FROM {table}];
```

In this example, we denote **keywords**, or basic statement components, in uppercase. Again, recall that SQL statements are case-insensitive and that the capitalization is not mandatory but is done for visual clarity. The keywords we see in this basic statement are `SELECT` and `FROM`. `SELECT` indicates that the statement will retrieve, or "select" data, while `FROM` specifies the table from which data will be retrieved. Within the braces ({}) are the lists of columns we select and the name of the table in question. In relational terminology, this action is known as **projection**; the act of projecting one or more columns from a table. The resulting projected data is sometimes referred to as a **dataset** or **rowset**. In a `SELECT` statement, our selection action can take one of the following forms:

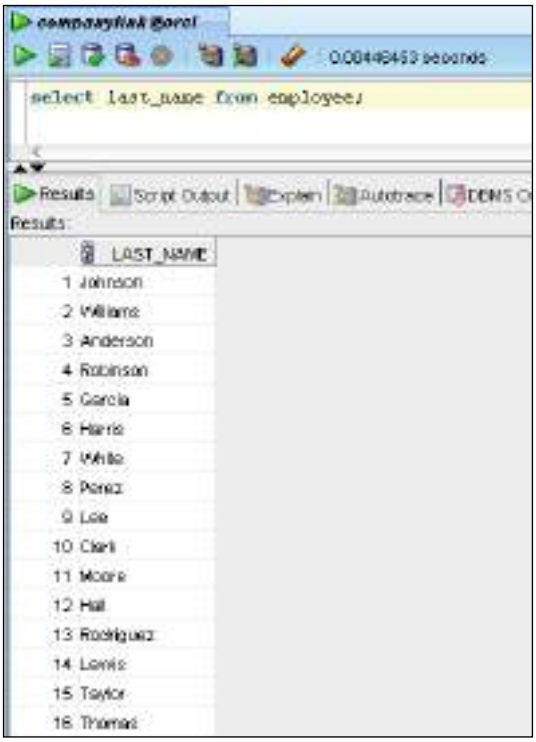
- Selecting one column from a table
- Selecting multiple columns from a table
- Selecting all columns from a table

Selecting a single column from a table

When we project a single column of data, our syntax tree takes the following form:

```
SELECT {column}
FROM {table};
```

In our example, `column` simply refers to the name of the column we wish to retrieve, and `table` refers to the name of the table. It is important to note that there is no row-based restriction in this statement—all rows in the table will be retrieved, but only of one column. Row restriction is a topic for a future chapter. The following screenshot contains an example `select` statement that you can type in and execute in SQL Developer. If we were to translate our `select` statement into everyday language, then it would be similar to *give me a list of all the employees' last names*.



We type the statement into the working-area frame in SQL Developer, click the execute button (see the previous chapter if you need a review of this), and the results are displayed in the results frame. In this example, the column name is `last_name`, and the table name is `employee`. The data in the `last_name` column is displayed exactly as it is stored in the table, without formatting, and in the same order that the records were entered.

SQL in the real world

It's important to see SQL statements not just as code, but to understand how they relate to the data they represent. Throughout this book, we will often translate our example statements into realistic language that we might better understand what our code is trying to accomplish.

Selecting multiple columns from a table

A multi-column projection of table data is very similar to a single-column projection. Its syntax tree is displayed as follows:

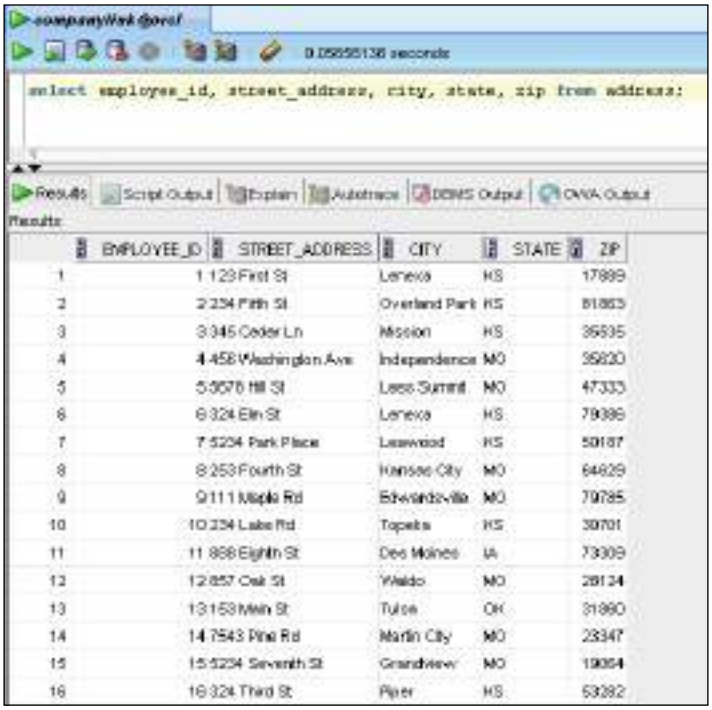
```
SELECT {column, column, …}
FROM {table};
```

We can use this statement format to project multiple columns of data from a single table. In our column list, we can identify two columns to retrieve or even more than two columns, as indicated by the ellipses (...). In fact, the maximum number of columns we can select is limited only by the number of columns in the table itself. The following screenshot shows an example from our `Companylink` database that makes the request *display the first and last names of all employees*:

As with our single-column select statement, this multi-column statement does nothing to restrict row data—all rows in the table are returned. Note also that the columns are shown in the order that they were requested—`first_name` followed by `last_name`. Even though the columns are stored in that order within the table, we are not restricted from selecting the columns out of order; in fact, that may often be desirable, as shown in the following screenshot. We might interpret it as, *display the last name of all employees, followed by their first name.*

As we see, the order in which the data is displayed is controlled by the statement we write, not by its order in the table. This applies to its position within the table as well. Notice that the first column in the `employee` table is `employee_id`, and yet that column was not selected or displayed at all. It is the writer of the SQL statement that determines what data will be displayed and in what order.

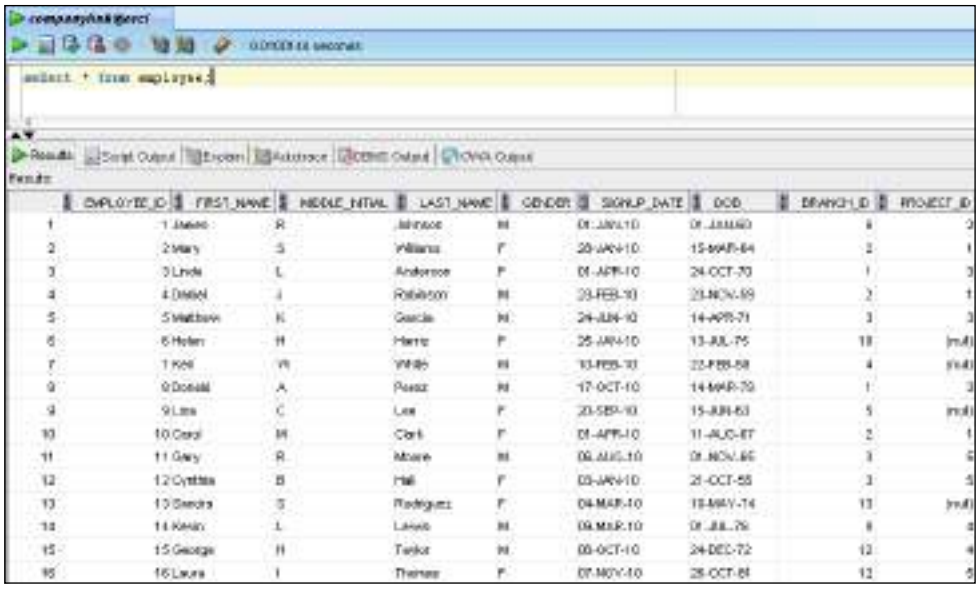
Another example is shown in the following screenshot. It uses the second, third, fourth, fifth, and sixth columns of a different table, the `address` table, but the order in which they are displayed is only controlled by the structure of the statement.



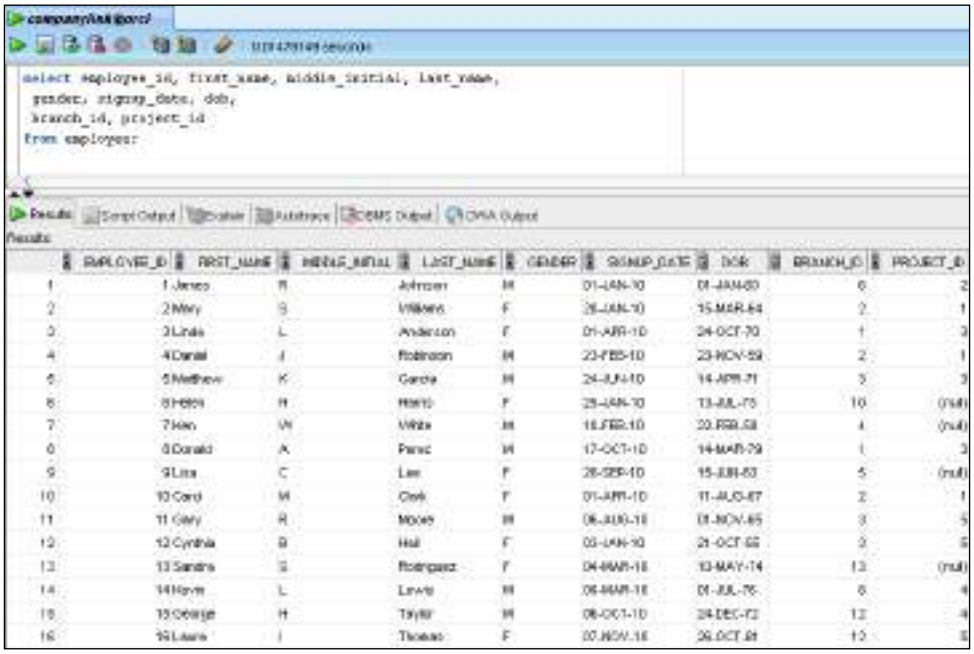
The real-life translation for this statement might be something like *give me employee ID and street address information*.

Selecting all columns from a table

The process of projecting all the columns in a table is relatively simple—it only requires the use of a special character. Recall, from our original syntax tree, that one of the options shown was an asterisk (`*`). When we use the asterisk in place of our column list, all columns in the table will be selected in the order that they appear in the table. The asterisk in an SQL statement is sometimes referred to as "star". You will sometimes hear the statement in the following example read as *select star from employee*.



Thus, all columns and all rows in the table will be displayed, since we've not yet seen a way to restrict the number of rows outputted. This statement, using `*`, would retrieve the same columns and in the same order as if we selected each of the columns individually, as shown in the following screenshot:



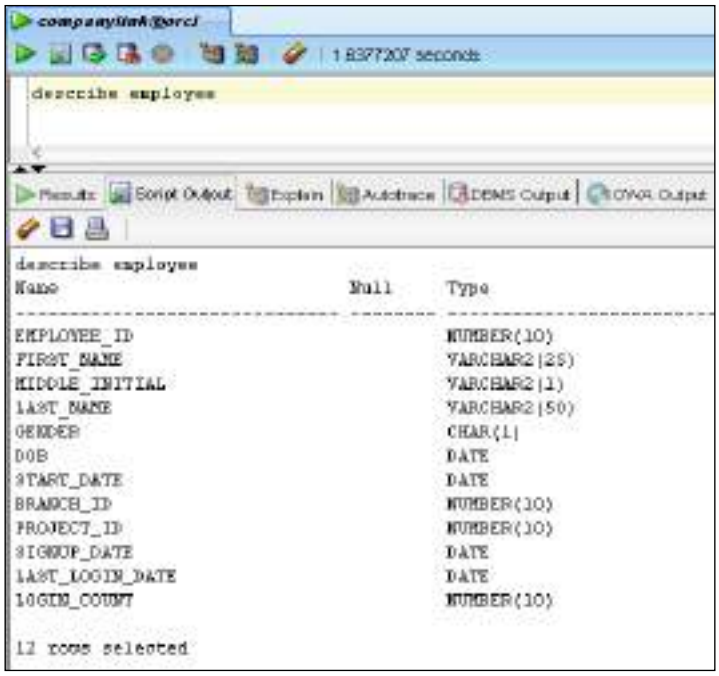
While either method achieves the same goal, remember that using `*` will only retrieve columns in the order that they exist in, in the table. Should you need to display all columns in a different order, it is necessary to select each column in the table by name.

SQL in the real world

In some IT shops, the coding standards of a particular group may preclude the use of `*` in SQL statements, and that's not necessarily a bad thing. While using the "star" notation requires less typing, it is also very non-descriptive for someone trying to read the code that you've written. Coding standards that mandate writing out each individual column produce code that is easier to read and debug.

Displaying the structure of a table using DESCRIBE

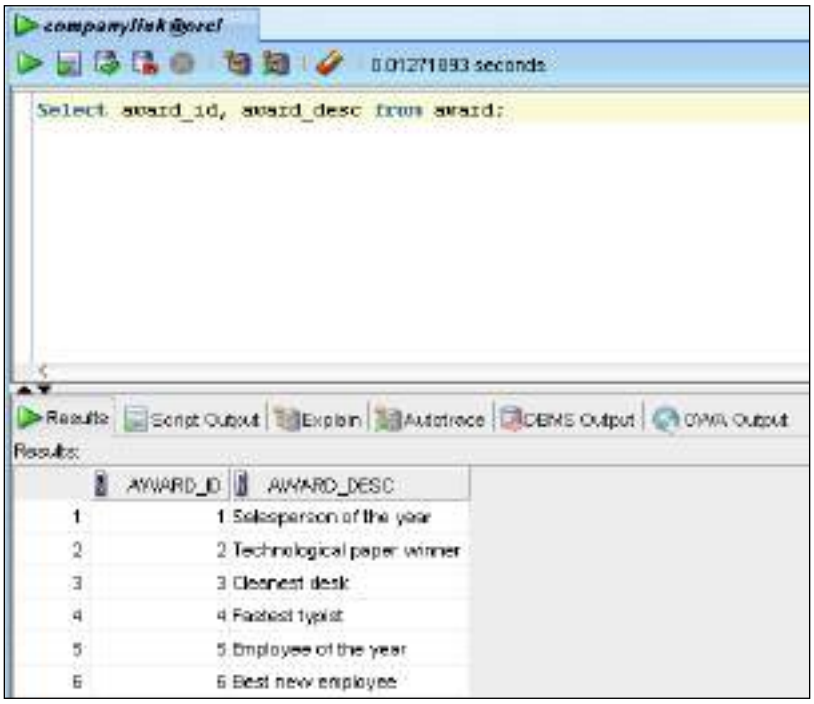
If, by now, in doing the examples in this chapter, you've received an error when incorrectly typing a column or table name (as is common), it should be obvious that SQL requires that we be syntactically accurate when listing each element of the statement. All clauses, keywords, column names, and table names must be spelled correctly. This can be difficult to do if you don't know what the column names are. Although our SQL Developer tool makes this easy to find, we also have the `DESCRIBE` command to display the structure of a table, as shown in the following screenshot:



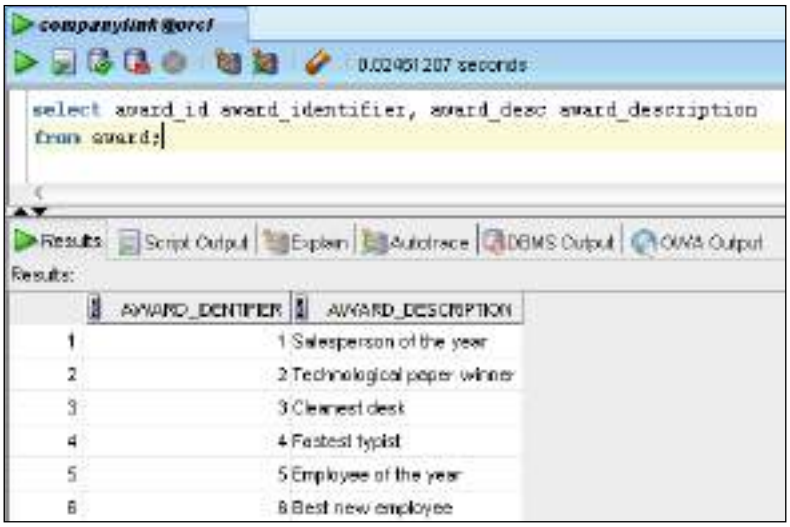
Executing the command produces three columns of information: the name of the column, whether it allows NULL values, and the datatype of the column. At this point, we will concern ourselves only with the first and third columns. The output shown indicates that our `employee_id` column is of type **NUMBER(10)**. We will examine datatypes further in later chapters; but, for now, this information tells us that the `employee_id` column contains numeric values and that other columns, such as `first_name` and `last_name`, contain character data, while columns such as `start_date` and `last_login_date` contain date data. Whether you use the `DESCRIBE` command or other tools to display column names, it is important to do so in order to write error-free SQL.

Using aliases to format output of SELECT statements

While we have demonstrated that we can retrieve data from our `Companylink` database, we have not, thus far, been able to alter the way the data is presented. As we move through the book, we will discover more ways to customize our output, but one way we can do this now is through the use of an alias. An **alias** is an alternate name given to a column that alters the way it is displayed. To demonstrate this, let's examine another one of our `Companylink` tables: the `award` table. The following example shows a query from the `award` table that selects its two columns but uses no alias:



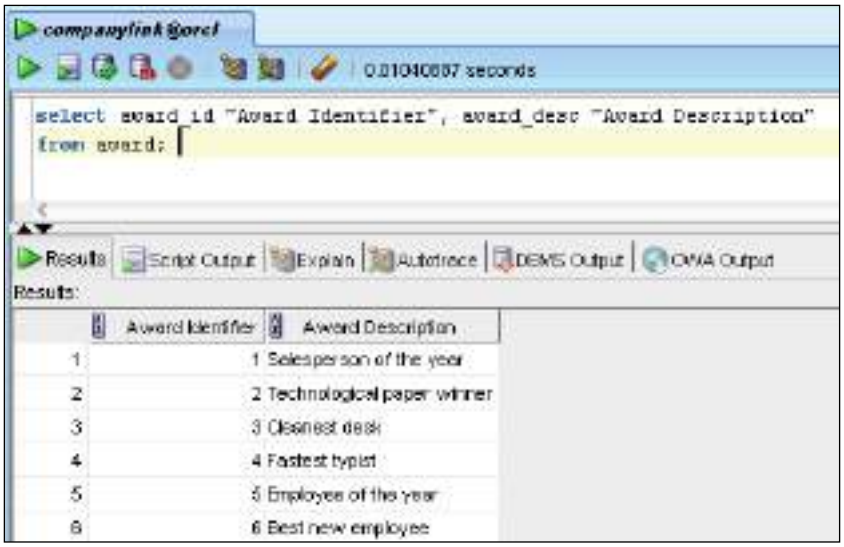
For the first time, we turn our attention to a portion of the output other than the data. Notice how the column headers, **AWARD_ID** and **AWARD_DESC**, are named exactly as the column names themselves. Of course, this is to be expected, but what if we wanted to change the column headers to something more descriptive? To do this, we simply use an alias for the columns, as shown in the following screenshot:



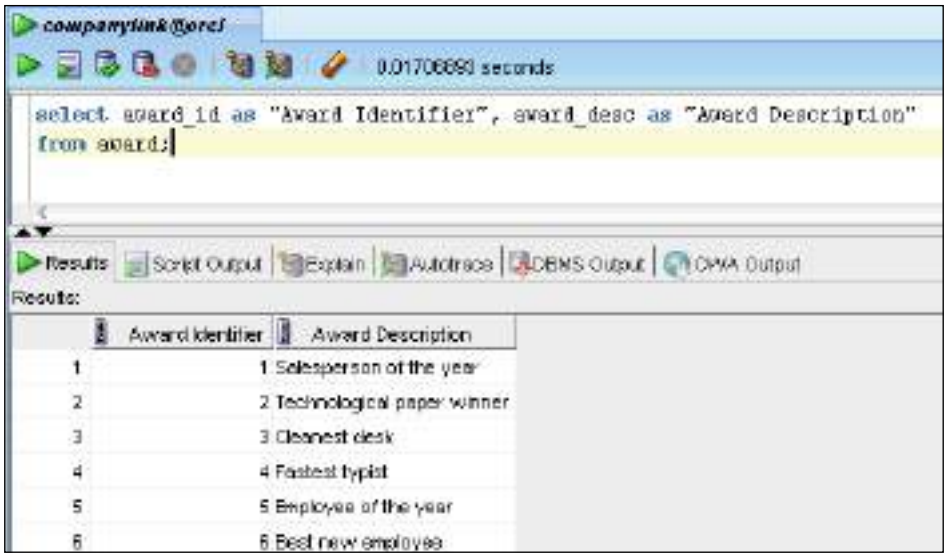
Notice how the column headings have changed to **AWARD_IDENTIFIER** and **AWARD_DESCRIPTION**, respectively. To accomplish this, we simply list the alias immediately following the column that it replaces. Thus, when we list **award_identifier** directly after the **award_id** column, the display name is changed. Note that this in no way means the actual name of the column **award_id** has changed within the table. It simply means that it is aliased, or displayed with a different name.

You may notice that, while we have managed to change the column heading to something different than the column name, we haven't changed it significantly. The alias still uses all uppercase characters and an underscore. We can, however, modify the column's appearance further. Doing so will require that we use an exception to two of the SQL rules we listed earlier in the chapter. Recall that in the earlier sections of this chapter, *Case sensitivity* and *The use of whitespace*, we stated that our SQL statements were case-insensitive and that whitespace, such as tabs and extra spaces, is ignored. There is one important exception to these rules, and we see a good example of it in the use of aliases. Case-sensitivity and whitespace in an SQL statement can be maintained if we enclose the alias name in double quotes (`""`).

We see an example of this in the following example (which uses both case-sensitivity and whitespace within an alias):



If we notice the column headings in the output after executing the previous statement, we see that the column headings for `award_id` and `award_desc` are now **"Award Identifier"** and **"Award Description"**, respectively, both of which use mixed case and a space character within the column heading. Note that this can only be done using double quotes. If we were to attempt the use of mixed case and a space between the two words used in the alias, without double quotes, the mixed case would be ignored, and we would receive an error because Oracle would not interpret the space character in the statement. For a neater overall appearance in our code, we can use the optional `AS` keyword to denote our alias, as indicated in the following screenshot:



SQL in the real world

Using aliases to format column headings was a more common practice in the past, when reports were run using SQL*Plus. This is less common today. In modern software development, data is formatted by the application and does not require the use of aliases. However, there are other important reasons to use aliases. When you alias a column, you can refer to the column using that aliased name throughout the entire `SELECT` statement. We will see some important uses for this in later chapters.

Using arithmetic operators with SELECT

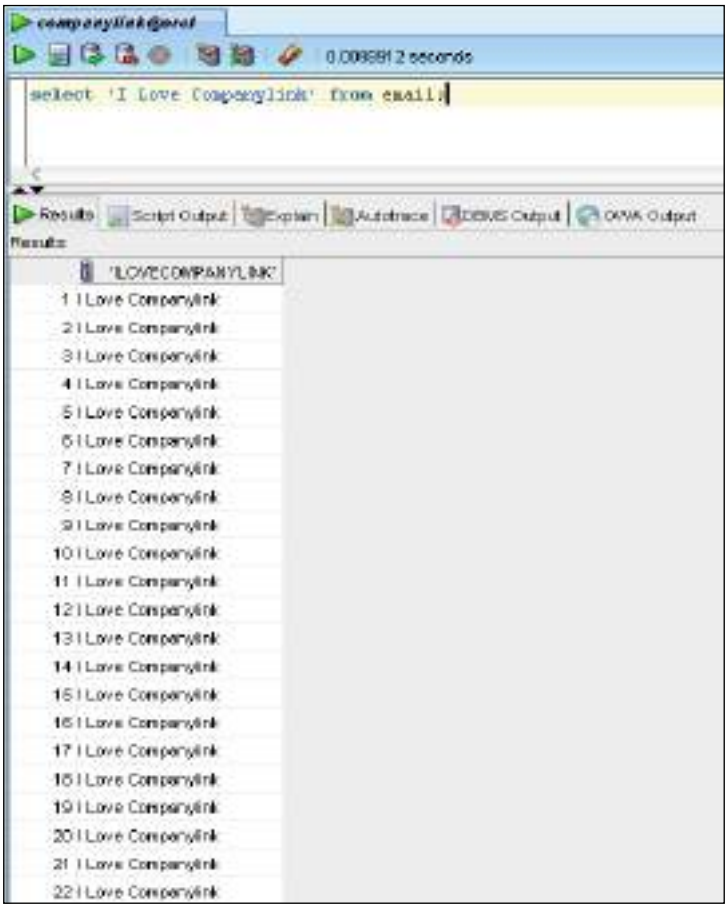
Although we've looked at several examples of using `SELECT` statements to manipulate strings of text, we can just as easily use SQL to complete arithmetic operations. As with many programming languages, we can make use of arithmetic operators to accomplish this.

The DUAL table and the use of string literals

Let's say, for a moment, you did not want to display data from a table, but rather a fixed literal statement such as "I Love Companylink". We need to introduce another set of syntactical operators, the single quotes (`'`), in order to do this. In SQL, when any word or phrase is enclosed within single quotes, it becomes expressed as a string literal. However, how would we display our literal statement using SQL? The only statement at our disposal, so far, is the `SELECT` statement. We could attempt a statement as follows:

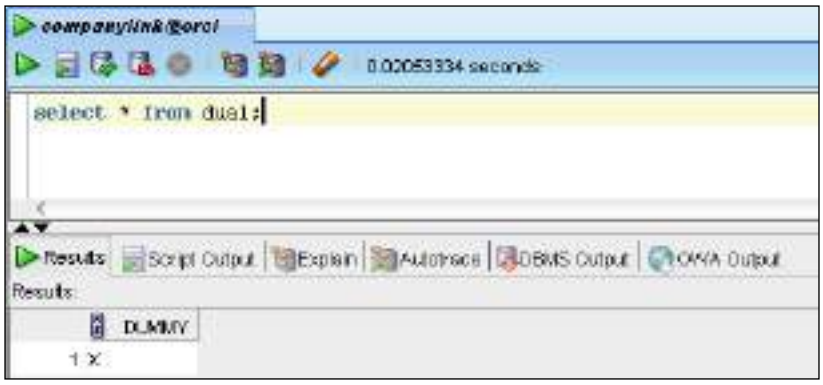
```
Select 'I Love Companylink';
```

However, such a statement is syntactically incorrect, since every `SELECT` statement requires a `FROM` clause; thus, we would need to select our literal from some table. However, notice what happens when we attempt this using the e-mail table, as shown in the following screenshot:



In this select statement, we haven't selected any actual columns, only a literal expression. The result is that one literal is returned for every row in the table, simply repeating over and over. What we need is a table with only one row to select against. This is why Oracle has provided the `DUAL` table.

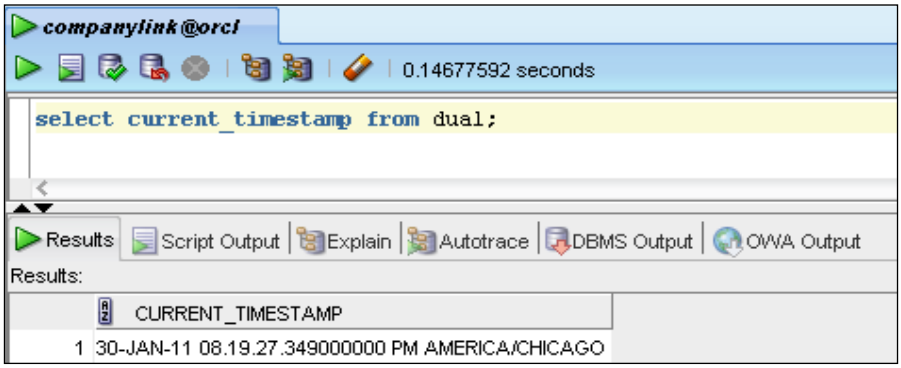
`DUAL` is essentially a **pseudo-table**—it has no real data and is generally used for string manipulation and mathematical computation. You do not insert, update, or delete data from the `DUAL` table. The following example shows the result of selecting from **DUAL**:



The only column in **dual** is **DUMMY,** and the only row value is **X**. However, because it only has one row, it becomes the perfect solution to our problem of how to select string literals without repeating values, as shown in the next screenshot:

One common use of the `DUAL` table is to display the current date and time. To do so, we make use of a **pseudo-column** in our select statement. For instance, we can use the `SYSDATE` pseudo-column to display the current date, as shown in the following screenshot:

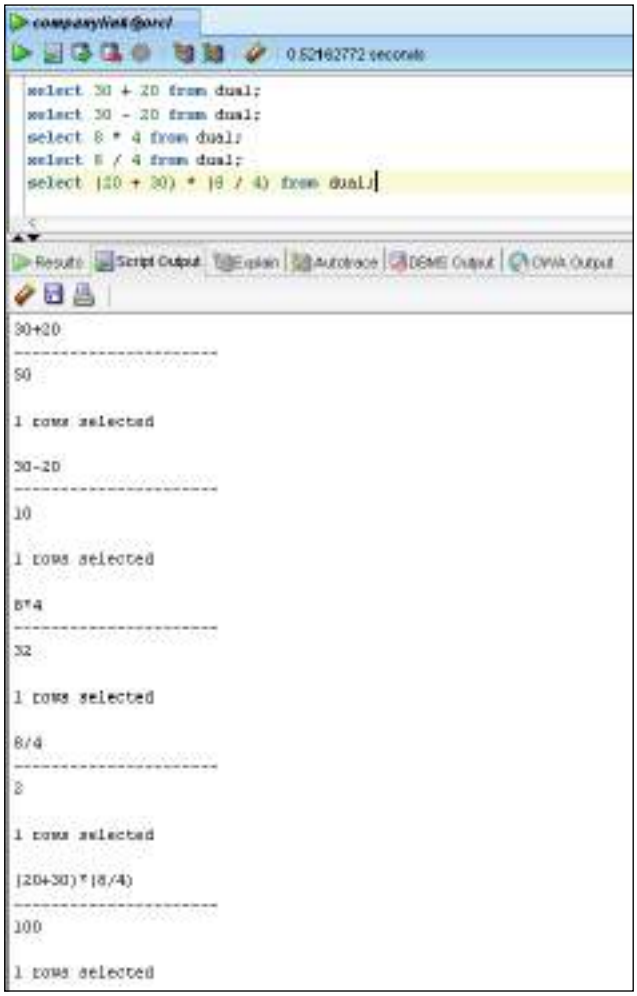
The value that is returned is the current system date. Oracle also provides the `current_timestamp` pseudo-column to retrieve date, time, and time zone information from the server, as shown in the following screenshot:



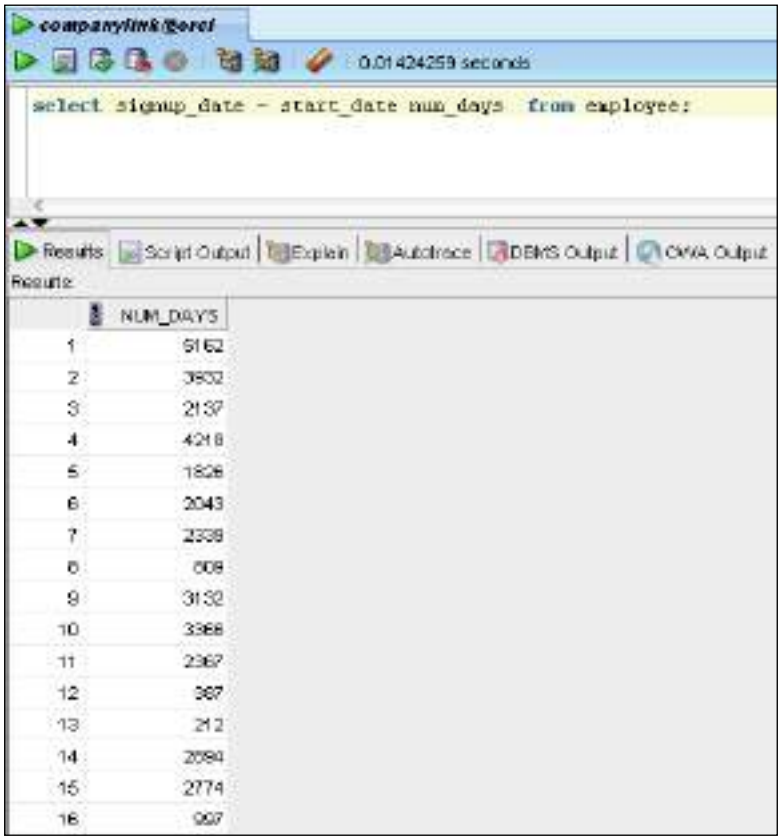
Note that the date and/or time presented by both of these pseudo-columns reflects the system date/time of the server on which the database is hosted. It may be different from the date/time of the actual client that executes the query.

Mathematical operators with SELECT

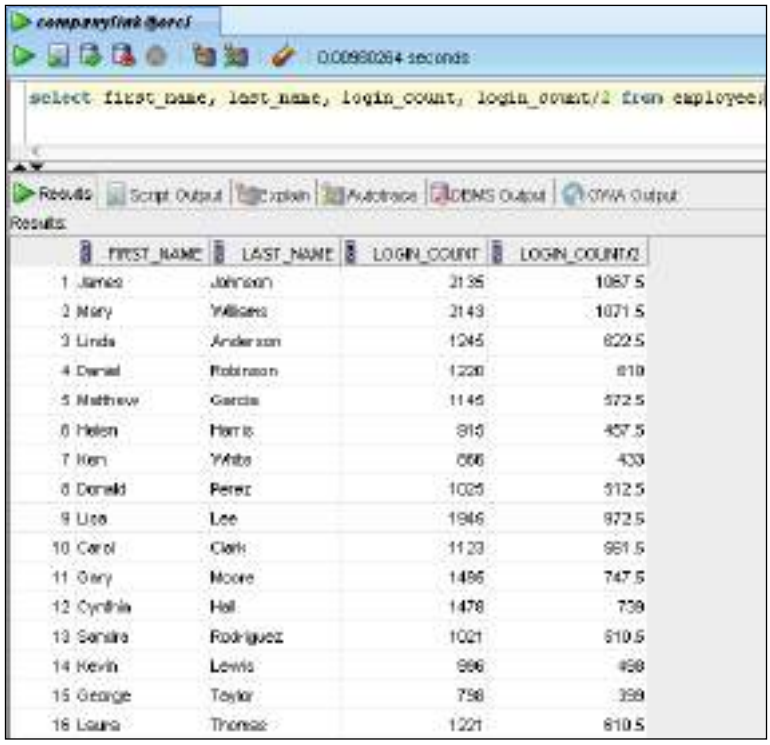
Our `Companylink` database will almost certainly require the use of mathematical computation in order to be effective. Let us suppose we want to calculate the number of days until the birthday of each of our employees or how long a particular employee has worked for the company. Basic mathematical operations are the heart of any computing system. In Oracle, many mathematical operations are accomplished using functions. We can, however, do basic mathematical operations within our `SELECT` statements. The following example shows some simple examples of this. In SQL Developer, in order to execute multiple statements, it is necessary to use the **Run Script** button located just to the right of the green **Run** button or, alternatively, to press the *F5* key.



As you can see from the example, basic mathematical operations such as addition, subtraction, multiplication, and division are symbolized by `+`, `-`, `*`, and `/`, respectively. The order of precedence is the same as in basic math, with parentheses taking precedence over multiplication and division, which take precedence over addition and subtraction. In SQL, the mathematical operands can be either literals or column data. In the previous example, the operands (the number **30** and the number **20**) are literals. The SQL statement takes the literal operands as input and "selects" their sum against the **dual** table, which returns one row: **30 + 20**. However, although mathematical operators work with literals, the real power in using mathematical operators with SQL is the ability to execute an operation on every row in a table. For instance, if you have a table containing one million rows, trying to multiply the values of two columns using each literal value would be fruitlessly time consuming. In SQL, you can write one statement that will recursively execute for every desired data element in the table. The following example demonstrates the use of columns in mathematical operations:



This SQL statement demonstrates date arithmetic. Two columns from the `employee` table, `start_date` and `signup_date`, are subtracted. The result of this operation is that the number of days between the `start_date` and the `signup_date` is returned for each row in the table and displayed in a column that is aliased as `num_days`. It is important to understand that the previous SQL statement does not simply execute the subtraction once; rather, it does it recursively for each row. Mathematical operations can also contain columns mixed with literals, as shown in the following screenshot:

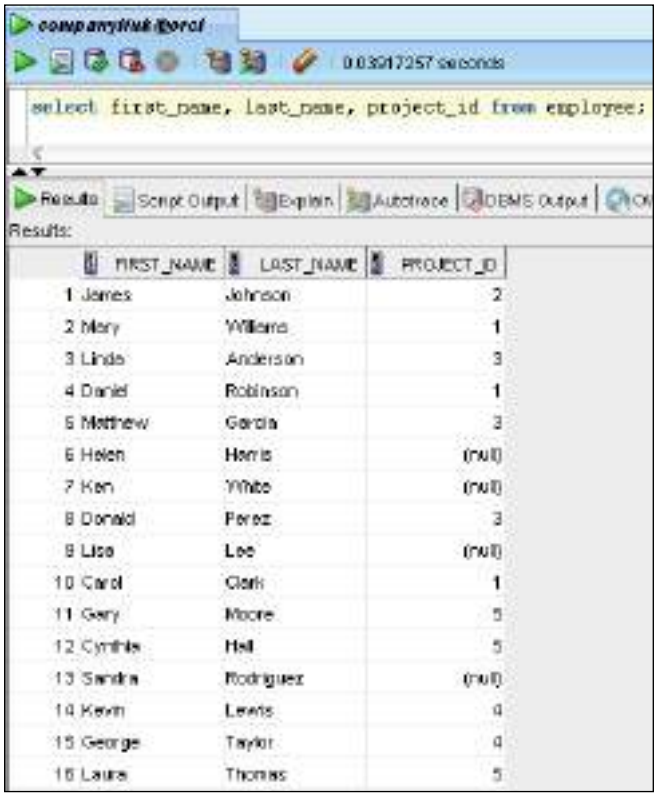


In this example, the `login_count` column contains the number of times an employee has logged in to *Companylink*. We divide this column value by two, for each row, and return the result. For the sake of clarity, we also select the columns `first_name`, `last_name`, and the unmodified `login_count` column.

Numeric values are allowed for any mathematical operation, while character values are not. Date values are allowed, but only when using subtraction, as our example shows.

The meaning of nothing

To complete our section on mathematical operators and SQL, we need to examine one final reserved keyword—`NULL`. In SQL, the `NULL` keyword signifies the lack of data. In short, NULL means nothing or "undefined". It should never be confused with the "space" or "zero" values—these values actually do constitute data. A space is a string character value. A zero is a true number. A NULL is neither. Although it is often referred to as a "null value", it is not data; rather, it is the absence of data. As such, it is treated differently than numeric, string, or date values. In later chapters, we will examine the rules of how to put NULL values into a table; but, for now, we need to learn how to recognize them when they are returned in queries. The following screenshot provides us an example of NULLs returned from a query:



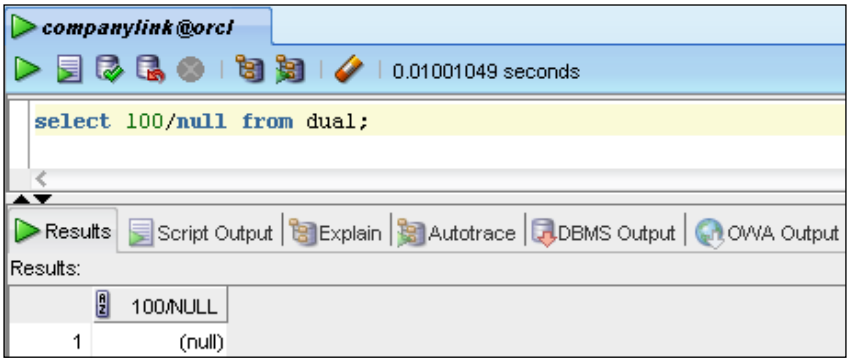
In the values returned, we see that some employees have a **PROJECT_ID** and some do not. **James Johnson** is associated with a **PROJECT_ID** of **2,** and **Mary Williams** has a **PROJECT_ID** of **1**. However, **Helen Harris**, **Ken White**, **Lisa Lee**, and **Sandra Rodriguez** all return a value of **(null)** for their **PROJECT_ID**. This is simply because there is no data value in the **PROJECT_ID** column for those employees.

SQL in the real world

In SQL Developer, a NULL value is shown as `(null)`. Other SQL tools may display NULLs differently. For instance, SQL*Plus simply displays NULL as whitespace, while Oracle's Application Express tool represents them with a dash. It's important to know how the tool of your choice renders NULLs so that you can recognize them.

To illustrate the fact that NULLs truly have no value, examine the following error box. In it, we attempt the statement `select 100/0 from dual`, which should produce an error, since division by zero constitutes a mathematical error. When we execute the statement, we receive the following error:

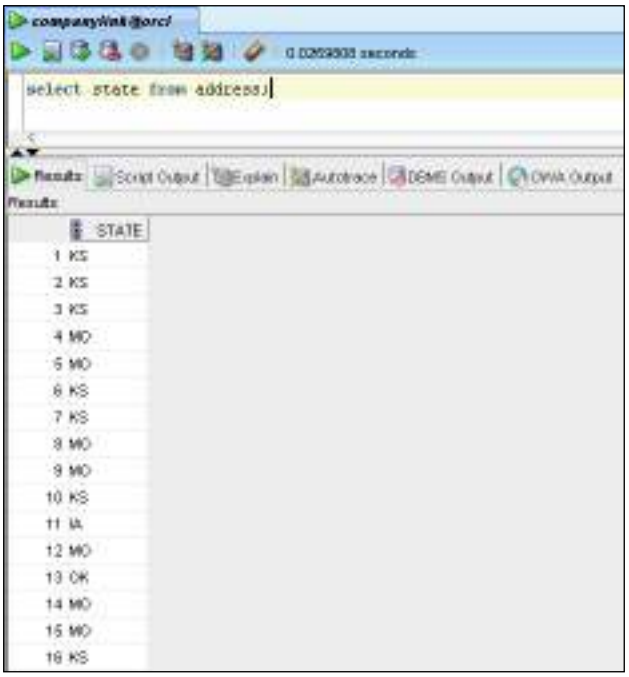
This is the expected behavior, since `100` is a numeric value. However, in the next example, we attempt a similar query using a **null** instead:



As we can see from the result, no error is raised. Instead, dividing 100 by NULL simply produces another NULL, indicating that NULL is not a value at all.

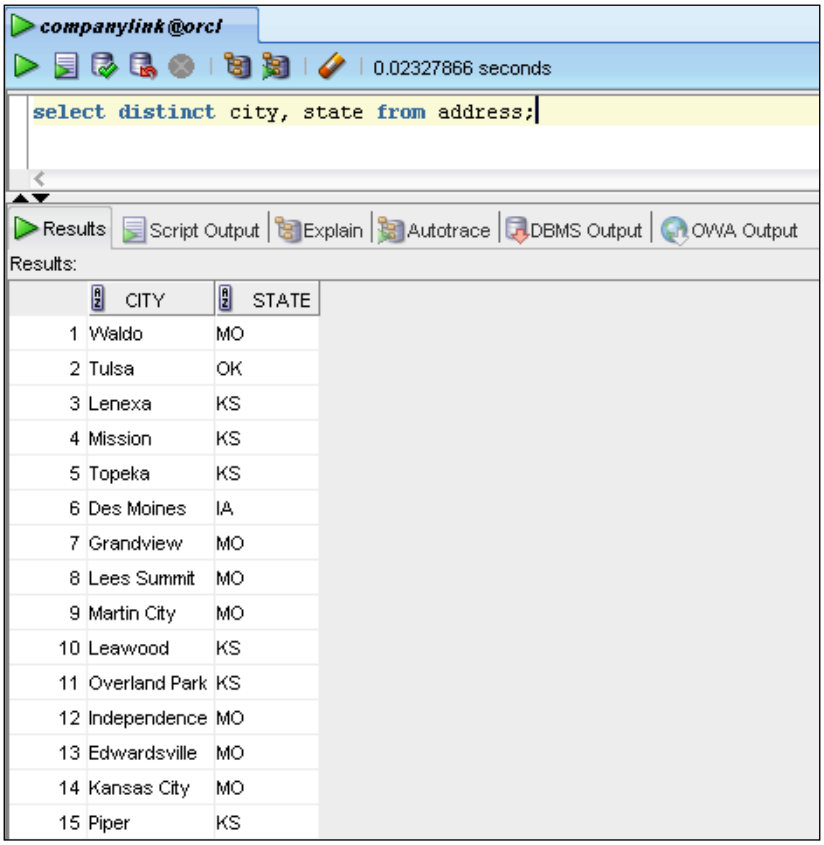
Using DISTINCT to display unique values

Consider the `address` table, which lists all of our employees' addresses. Let us suppose that we want a report on which states our employees are from, to satisfy federal tax guidelines. We could do this using the following query:



The results give us what we asked for, but they are littered with duplicate values. If we want a concise list of states where our employees live, it would be preferable to discard duplicate values and display only a unique list of states. We can do this using the **distinct** keyword, as shown in the following example:

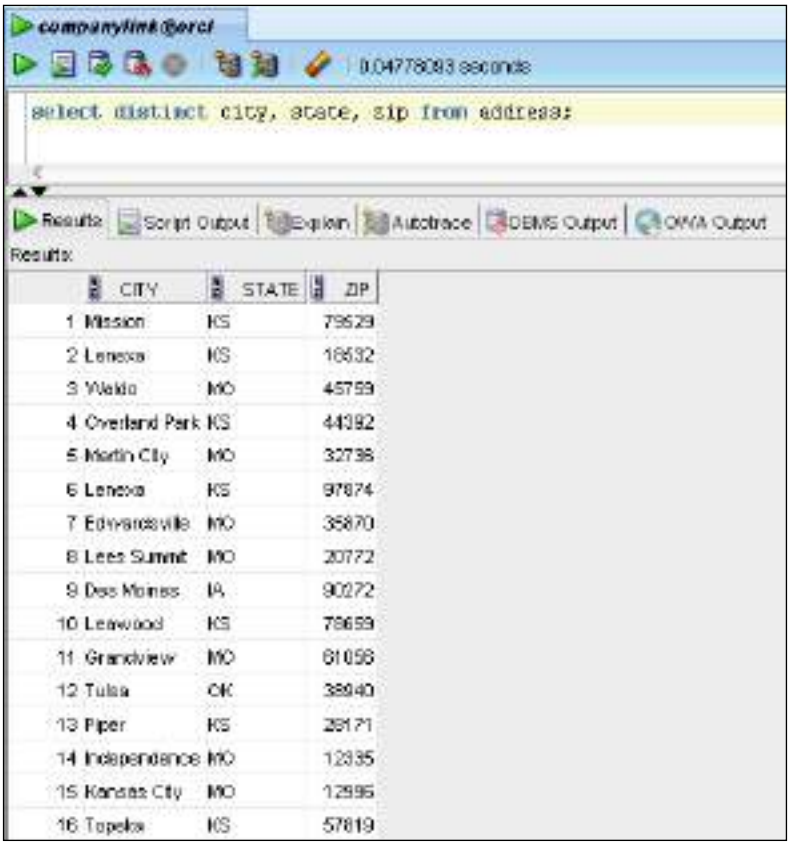
While the original query returned 16 rows, the query using `DISTINCT` removed all redundant values and returns only four rows. Another powerful use of the `DISTINCT` keyword is the ability to return distinct values from multi-column datasets. For example, the following SQL statement demonstrates how `DISTINCT` operates on multiple columns of data:



	CITY	STATE
1	Waldo	MO
2	Tulsa	OK
3	Lenexa	KS
4	Mission	KS
5	Topeka	KS
6	Des Moines	IA
7	Grandview	MO
8	Lees Summit	MO
9	Martin City	MO
10	Leawood	KS
11	Overland Park	KS
12	Independence	MO
13	Edwardsville	MO
14	Kansas City	MO
15	Piper	KS

While a query that omits the `DISTINCT` keyword will return 16 rows, this query returns only 15, since there are two occurrences of the city/state combination **Lenexa Kansas**, and one of them is removed by the action performed by the `DISTINCT` keyword.

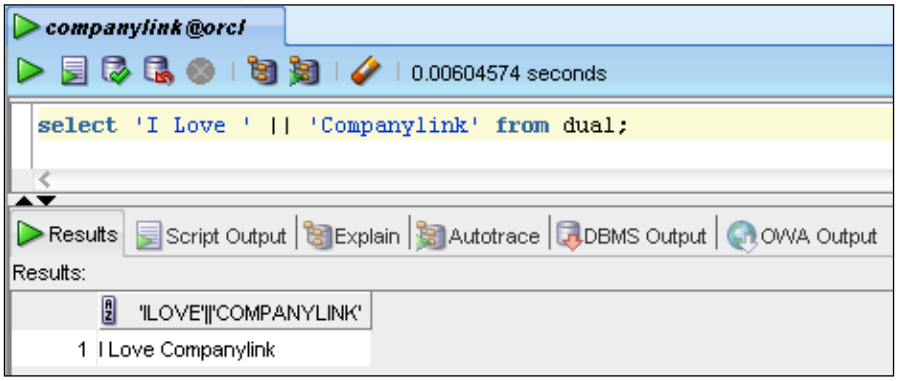
It is interesting to note, however, that if we add the `ZIP` column to the query, the number of rows returned is 16. Observe this in the following screenshot:



Why does this occur? Look closely at the data, and you will see that even though the combination of **Lenexa** and **Kansas** is repeated, each occurrence of it has a different value for the `ZIP` column. This results in three values that, when taken together, are fully distinct.

Concatenating values in SELECT statements

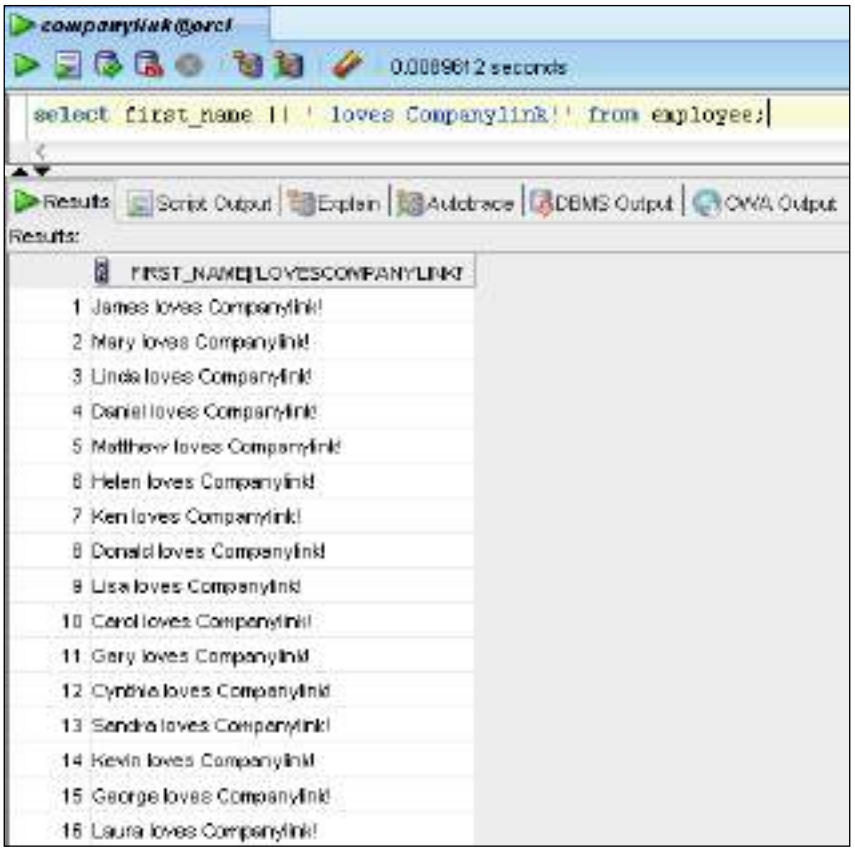
While SQL allows us to manipulate numeric values with mathematical operators, we can also do some degree of character string manipulation with our statements. Earlier in the chapter, we used the `DUAL` table to display string literals that were enclosed in single quotes. We can use concatenation to append one or more string literals with values from a table. There are actually two ways we can concatenate values in Oracle, but, in this section, we examine the concatenation operator "double-pipe" or (||). The pipe symbol is invoked on most keyboards using *Shift* + \ (backslash). We use two of these pipe symbols or "double pipe" to concatenate values in SQL. A basic example of concatenation using the **dual** table is shown in the following example:



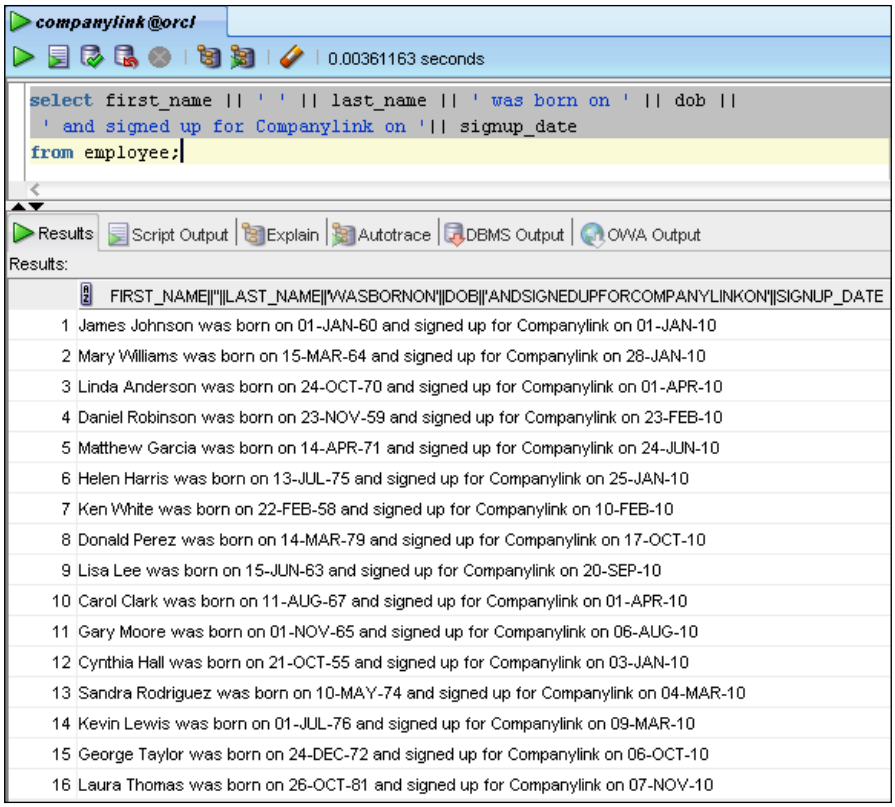
You can see that the first string, **'I Love '**, is concatenated with the second, **'Companylink'**, and displayed together as **'I Love Companylink'**. It is important to notice two things in this statement, aside from the use of the double pipe. First, remember that, in this example, we are using string literals, so they are surrounded by single quote marks and **not** double quotes. Recall from earlier in the chapter that double quotes are used for aliases. Because we use single quotes, everything inside them is interpreted as a string literal, including the spaces. So, in the string **'I Love '**, the second space is very important. Were we to omit it, the resulting string from the execution of the SQL statement would be '*I LoveCompanylink*'.

Alternatively, you could place the space before the **'Companylink'** string as ' *Companylink*' and achieve the same result.

The opportunity provided by using concatenation is better seen when coupled with actual values from a database table. Using the double pipe, we can combine table results with string literals of our choice to produce formatted output, as shown in the following screenshot:



In this example, we select the `first_name` column from the `employee` table and append the string literal **' loves Companylink!'** to it. The statement executes this append operation for every row and returns it as the rowset you see in the results. A more involved, as well as practical, example is shown in the following screenshot:

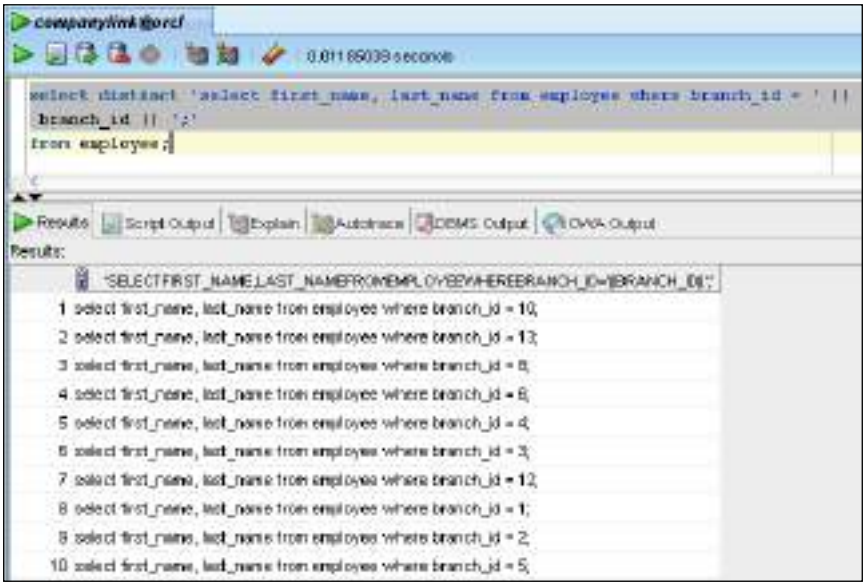


	FIRST_NAME||' '||LAST_NAME||'WASBORNON'||DOB||'ANDSIGNEDUPFORCOMPANYLINKON'||SIGNUP_DATE
1	James Johnson was born on 01-JAN-60 and signed up for Companylink on 01-JAN-10
2	Mary Williams was born on 15-MAR-64 and signed up for Companylink on 28-JAN-10
3	Linda Anderson was born on 24-OCT-70 and signed up for Companylink on 01-APR-10
4	Daniel Robinson was born on 23-NOV-59 and signed up for Companylink on 23-FEB-10
5	Matthew Garcia was born on 14-APR-71 and signed up for Companylink on 24-JUN-10
6	Helen Harris was born on 13-JUL-75 and signed up for Companylink on 25-JAN-10
7	Ken White was born on 22-FEB-58 and signed up for Companylink on 10-FEB-10
8	Donald Perez was born on 14-MAR-79 and signed up for Companylink on 17-OCT-10
9	Lisa Lee was born on 15-JUN-63 and signed up for Companylink on 20-SEP-10
10	Carol Clark was born on 11-AUG-67 and signed up for Companylink on 01-APR-10
11	Gary Moore was born on 01-NOV-65 and signed up for Companylink on 06-AUG-10
12	Cynthia Hall was born on 21-OCT-55 and signed up for Companylink on 03-JAN-10
13	Sandra Rodriguez was born on 10-MAY-74 and signed up for Companylink on 04-MAR-10
14	Kevin Lewis was born on 01-JUL-76 and signed up for Companylink on 09-MAR-10
15	George Taylor was born on 24-DEC-72 and signed up for Companylink on 06-OCT-10
16	Laura Thomas was born on 26-OCT-81 and signed up for Companylink on 07-NOV-10

It is easy to be overwhelmed with the syntax in the previous statement, but understanding it is simply a matter of breaking it down into pieces. Remember that these statements only execute according to the rules of SQL. Let's break down the steps of execution for the statement:

1. Select the **first_name** column.
2. Append a space ' ' string literal to it.
3. Append the **last_name** column.
4. Append the string **' was born on '**.
5. Append the **dob** column.
6. Append the string **' and signed up for Companylink on '**.
7. Append the column **signup_date**.
8. Specify which table the column data comes from, namely the **employee** table.

If you encounter an error when you execute your statement, go through the syntax, one step at a time. One common mistake is to have a beginning single quote without an ending one. Make sure you identify which elements are string literals and which are table columns. Once we understand the rules of concatenation, the possibilities are endless. The following screenshot demonstrates an advanced example of how DBAs use concatenation to construct SQL statements that can be executed separately:



The results from this statement are completely different from those in previous examples. In this case, our results are not data, per se, but are actually other SQL statements. We have formed these statements using the values from the **employee** table; therefore, they can differ each time we execute the statement, based on the values for the `employee` table at that time. Once these statements are generated, we can simply copy and paste them into SQL Developer and execute them one at a time, as shown in the following screenshot:

You may also have noticed that we have introduced a new clause in our statement: the **WHERE** clause. This is the subject of our next chapter.

SQL in the real world

The Oracle RDBMS has a special set of tables known as the data dictionary, which contains a massive amount of metadata about the database itself, such as table and column names. It is common for DBAs to use statements similar to the previous one to leverage the data dictionary in generating their own dynamic SQL statements. This can also be done in a different way in the PL/SQL programming language, using a feature called **NDS**, or **Native Dynamic SQL**.

Summary

In this chapter, we've introduced our first SQL statement, the `SELECT` statement, examined its syntax, and explored how it can be used in the process of column projection. We've seen the use of the `SELECT` statement in projecting single columns and multiple columns. We have looked at the `DESCRIBE` command and how it can be used to display the column names for a table. We've covered the use of aliases in changing column headings and used arithmetic operators to execute mathematical calculations on table data. We've examined the concept of NULL and demonstrated how to retrieve unique rowsets using `DISTINCT`. Finally, we've used the double pipe symbol to concatenate string literals with table values.

Certification objectives covered

- Listed the capabilities of SQL `SELECT` statements
- Executed a basic `SELECT` statement

So far, we've focused on the topic of projection in the SQL language, which is the process of restricting certain columns of data and displaying them in the manner we wish. The next chapter focuses on the topic of selection, or restricting our retrieved rowsets to certain rows of data, using the `WHERE` clause in SQL.

Test your knowledge

1. What is the name given to the set of rules that define a programming language's structures, symbols, and semantics?
   a. Projection
   b. Restriction
   c. Syntax
   d. If..then
2. In which of these statements is case preserved?
   a. SElect * from employee;
   b. select first_name "My Name" from employee;
   c. SELECT BLOG_ID FROM BLOG;
   d. SelECt diSTINCT branch_ID, BRANCH_name from branch;
3. Which of these symbols is a valid statement terminator in Oracle SQL?
   a. !
   b. *
   c. ;
   d. &
4. Which of these is not a valid SQL statement?
   a. select first_name from employee;
   b. select * from message;
   c. select from award_date award;
   d. select website_url from website;
5. The process of displaying one or more columns from a table is known as:
   a. Projection
   b. DISTINCT
   c. Restriction
   d. Selection
6. Which of these is not a valid SELECT statement (refer to the Companylink tables, if needed)?
   a. select LAST_NAME from employee;
   b. select project_description from project;

c. select employee_id from email;

d. select address_ id from address;

7. Which of these is not a valid SELECT statement (refer to the Companylink tables, if needed)?
   a. select first_name last_name from employee;
   b. select employee_id, website_id from website;
   c. Select DIVISION_id, division_name from division;
   d. select blog_id, blog_web_url from blog;

8. Which of these statements will satisfy the request "display the first name, last name, and gender of all the employees in the EMPLOYEE table"?
   a. select first_name last_name, gender from employee;
   b. select first_name, last_name, gender from employee;
   c. select first_name, last_name, dob from employee;
   d. select first_name, last_name gender from employee;

9. Which special character is used in SQL to display all columns in a table?
   a. *
   b. ||
   c. $
   d. There is no special character to display all columns. You must list them individually.

10. What command is used in SQL to display the column names of a table?
    a. INSERT
    b. SYSDATE
    c. DISTINCT
    d. DESCRIBE

11. Which of these statements does not make valid use of an alias?
    a. select last_name "Last Name" from employee;
    b. select email_address 'My Email' from email;
    c. select distinct hit_count as "Hit Count" from website;
    d. select dob "Date of Birth" from employee;

12. Which of these statements will produce a column header of "Blog Description"?
    a. select blog_desc "BLOG_DESCRIPTION" from blog;
    b. select blog_desc 'Blog_Description' from blog;
    c. select blog_description from blog;
    d. select blog_desc as "Blog Description" from blog;

13. What is the output of the following statement?

```
Select 'Companylink is Very useful' from dual;
```

    a. Companylink is very useful
    b. Companylink Is Very Useful
    c. Companylink is Very useful
    d. companylink is very useful

14. Which of the following statements could be used to display the current date on the database server (choose all that apply)?
    a. select dual from dual;
    b. select sysdate from dual;
    c. select current_timestamp from dual;
    d. select current_sysdate from dual;
    e. select timestamp from dual;

15. Which of the following is not a valid use of mathematical operators in SQL?
    a. select 20 * 2 from dual;
    b. select start_date – signup_date from employee;
    c. select employee_id * street_address from address;
    d. select hit_count + 14 from website;

16. Which of the following statements will produce an error?
    a. select null from employee;
    b. select null from dual;
    c. select 50/null from dual;
    d. None of the above

17. Which keyword is used in SQL to discard duplicate values?
    a. DISCARD
    b. DISTINCT
    c. DUPLICATE
    d. SYSDATE

18. Which of these statements will produce an error?
    a. select ' ' || ' ' || award_desc || ' ' from award;
    b. select first_name || ' ' || last_name || 'is a great employee" from employee;
    c. select 'My project id is' || project_id || 'id' from project;
    d. select 'Award ID# ' || award_id || ' was presented on ' || date_ awarded || 'to employee' || employee_id from employee_award;

19. What is the output of the following statement?

```
select 'All employee' || 's should rem ' || 'member to ' ||
'return their' || 'badges' from dual;?
```

    a. All employees should remember to return their badges
    b. All employees should rem member to return their badges
    c. All employees should rem member to return theirbadges
    d. None of the above. The statement returns an error.

20. Which of these statements will produce the following result: Always remember your spaces when using concatenation?
    a. select 'Always remember ' || 'your spaces wh' || 'en using con' || 'cat' || 'enation'
    b. select 'Always remember ' || 'your spaces' || 'when using conc' || 'at' || 'e' || 'nation' from dual;
    c. select 'Always remember ' || 'your spaces wh' || 'en using con' || 'cat' || 'enation' from dual;
    d. select 'Always remember ' || 'your spaces when ' || ' using conc' || ' at ' || 'enation' from dual;

3
Using Conditional Statements

In *Chapter 2*, *Select Statements*, we worked with various forms of **projection**, in which `SELECT` statements allowed us to choose the columns we want to display. In this chapter, we explore the concept of **selection**, which allows us to limit our datasets to display only data that meets a certain set of conditions. Much of the chapter is spent examining the different types of conditions we can set. We will also learn the ways in which SQL can sort our data using the `ORDER BY` clause. By the end of this chapter, we will use both projection and selection to retrieve very specific sets of data and sort them in the order we wish.

In this chapter, we will cover the following topics:

- Examining the concept of data selection
- Understanding the structure and syntax of the `WHERE` clause
- Writing selective SQL statements with equality conditions
- Writing selective SQL statements with non-equality conditions
- Examining range conditions with the `BETWEEN` clause
- Examining set conditions using the `IN` clause
- Developing statements to do pattern-matching with the `LIKE` clause
- Understanding Boolean conditions in `WHERE` clauses
- Examining the use of the ampersand special character in substitution
- Exploring the sorting of data sets
- Examining the structure and syntax of the `ORDER BY` clause
- Understanding how to change the order of sorts using `ASC` and `DESC`

Implementing selectivity using the WHERE clause

Much of our ability to manipulate the potentially large amount of data in a database depends on restricting our data sets to exactly the data elements that we want to use. A data set that is too small, prevents us from accomplishing our task, and one that is too large can overwhelm our process from a performance standpoint. We now examine how we can use a new clause to selectively return rows based on a condition.

Understanding the concept of selectivity

In most of the examples we used in *Chapter 2*, every row was returned from each one of our queries. Only when using the `DISTINCT` clause were we able to restrict output to unique rows, and only if there were duplicates. If we projected the `first_name` and `last_name` columns from the `employee` table, every employee's first name and last name was returned. Until now, we had no way of displaying only the data that met a certain condition. For instance, say that we are working with our `employee` table from the `Companylink` database. We want to display `first_name` and `last_name` from the `employee` table, but we're really only interested in those employees who were born before 1979. Using the tools we learned in *Chapter 2*, we could use this statement:

```
..select first_name, last_name, dob from employee;
```

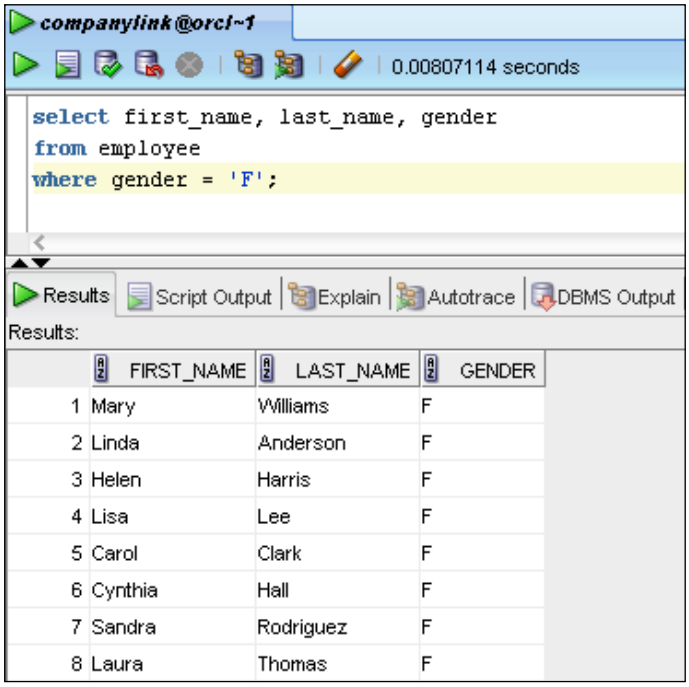
However, even though the data we need is retrieved, we are faced with the prospect of looking at each individual row and determining if the `dob` listed in the row meets our criteria. For 16 rows in a table, this is inconvenient; for a table with a million employee records, it is completely unworkable. Fortunately, the SQL language provides us with a way to restrict data sets based on one or more conditions—the `WHERE` clause.

Understanding the syntax of the WHERE clause

The syntax tree for a basic `SELECT` statement using a `WHERE` clause is shown in the following syntax tree:

```
SELECT {column, column, ...}
FROM {table}
WHERE {condition};
```

Even though we haven't yet given a specific example, we can notice a few things from the previous syntax tree. First, the basic forms of our `SELECT` and `FROM` clauses have not changed. We still select a column or columns, and we still specify our table name after the `FROM` clause. Second, we notice the position of our `WHERE` clause. It always follows the `FROM` clause, which follows the `SELECT` clause. Thus, our conditional statements follow the form `SELECT`, `FROM`, and `WHERE`. The condition itself takes the form of '*column name-operator-value or expression*'. The column name simply refers to the name of the column that forms the condition. The operator is often a mathematical operator, such as `=` or `>`. The value is a static value, such as the letter `F`, or in some cases, a variable. If the value is a character or date value, it must be enclosed in single quotes. An expression can also be used instead of a static value. We will see many examples of this throughout future chapters to help us better understand conditional statements. An example using data from the `Companylink` database is shown in the following screenshot. If we were to read the preceding statement as natural language, it might sound something like, '*Display the first name, last name, and gender of every female employee*'.

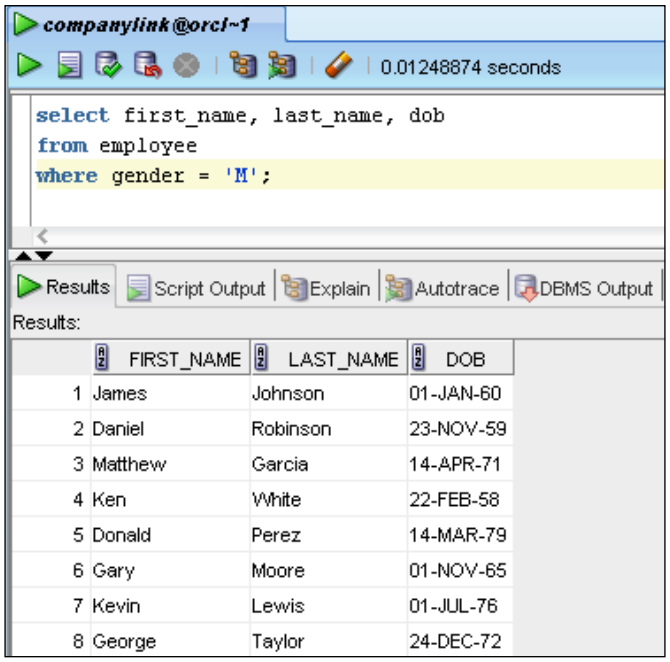


	FIRST_NAME	LAST_NAME	GENDER
1	Mary	Williams	F
2	Linda	Anderson	F
3	Helen	Harris	F
4	Lisa	Lee	F
5	Carol	Clark	F
6	Cynthia	Hall	F
7	Sandra	Rodriguez	F
8	Laura	Thomas	F

Using the `WHERE` clause, we restrict the data returned to a certain condition; namely, that the value in the `gender` column must be equivalent to the string literal value `F`.

How does this work? When the previously-listed statement is executed, the RDBMS examines each row individually and evaluates the value for the `gender` column. If the value equals `F`, the row is shown; if the value does not equal `F`, it is not shown. Nothing about the row changes in the table itself; it simply is not displayed.

You are not required to project, or include, the column that is evaluated in the condition with your selected column list. The following screenshot shows an example of this:



Notice in our example that although we're evaluating the values in the `gender` column, that column is not included in our selected list. We do not need to include the `gender` column in order for the statement to be syntactically correct. It is only required that the column being evaluated be a legitimate column in the `employee` table.

The examples we've shown so far all have a similarity in their `WHERE` clause—they're all examples of equality conditions. Each statement requires that a column value must be equivalent to a given string literal in order for the row to be returned. An equality condition is not the only type of condition that can be used to restrict rows in a `WHERE` clause. In the next section, we examine the different types of conditions that can be used.

SQL in the real world

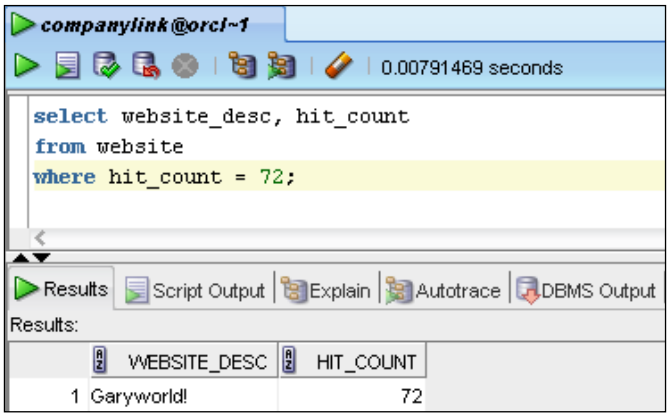
Remember that our goal in using the `WHERE` clause is row reduction; that is, reducing the number of rows that are returned to exactly the ones that meet our specified conditions. Our success at this will be determined by how well we utilize the various types of conditions at our disposal. Failure to do so in real situations can cause us to return more or fewer values than we intend.

Using conditions in WHERE clauses

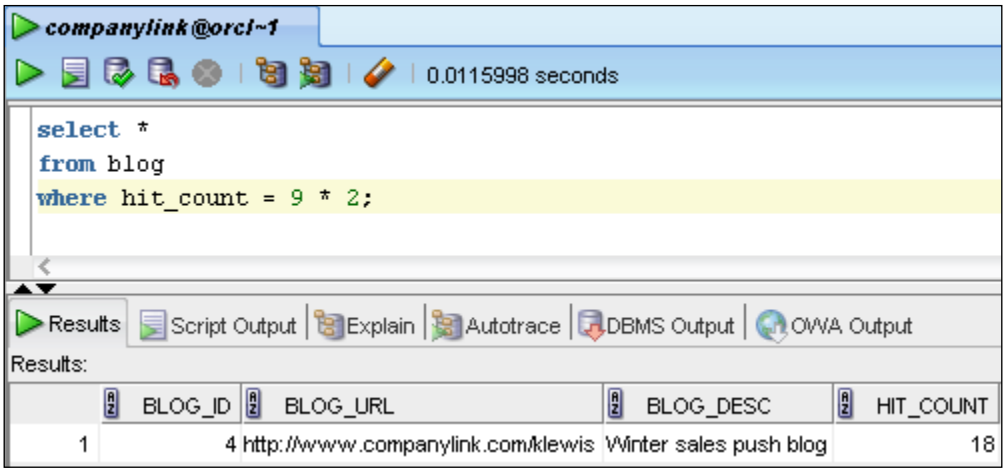
As we've mentioned, the previous examples in the chapter make use of equality conditions. That is, each column value must be equivalent to a given condition in order for the row to be returned. However, in these examples, the equality conditions given were only evaluated against string literals. Equality conditions can apply to numeric and date values as well as character strings.

Using equality conditions

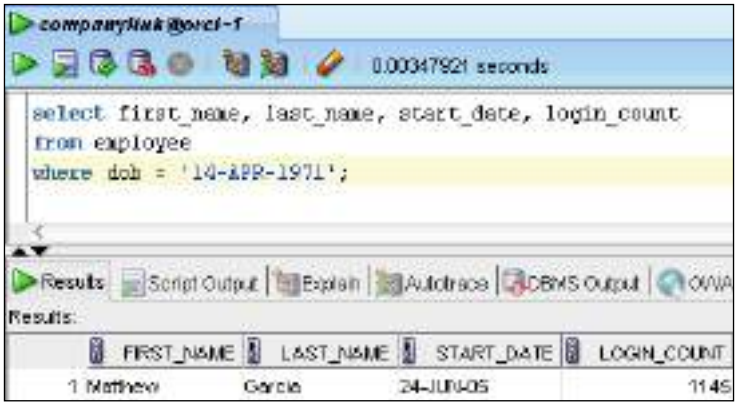
The following screenshot demonstrates an example of an equality condition:



The statement in the previous screenshot could be stated as, '*Show me the website and number of hits for any sites with 72 hits*'. It demonstrates the same syntax and use of the `WHERE` clause as used with string literals. We can also incorporate arithmetic expressions in our numeric conditions as well, as shown in the following screenshot:



The SQL statement interprets the condition 9 times 2 as 18 and scans the `blog` table for matching records. It finds the row for where `hit_count` is `18` and returns the entire row, as we used the asterisk symbol (`*`). Any of the mathematical symbols demonstrated in *Chapter 2*, will serve as a condition for the `WHERE` clause. Similarly, the `WHERE` clause can be used to restrict rows that meet a date condition as well. See the following screenshot that contains date conditions in a `WHERE` clause:

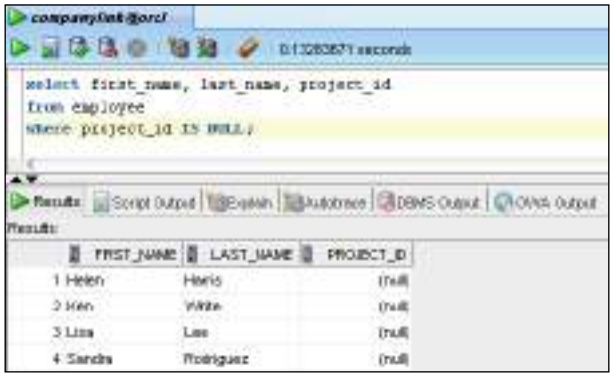


In this statement, we query for the employee's first name, last name, start date, and their login count, where the employee's date of birth is April 14, 1971. Oracle scans the `employee` table for dates in the `dob` column that meet this condition and returns one row.

SQL in the real world

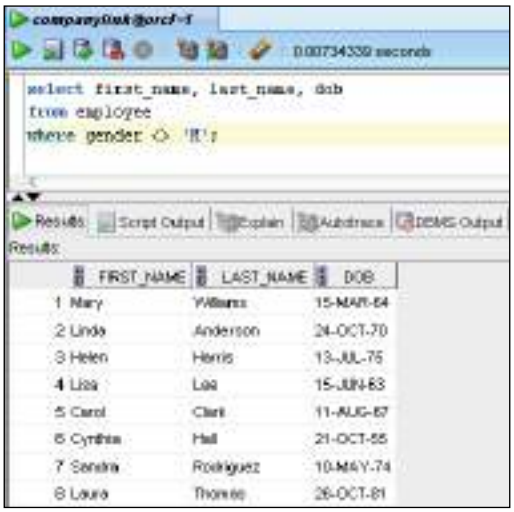
In real-life SQL statements, dates are often expressed differently, and Oracle does not always interpret them the same way. For instance, to Oracle, a date of April 14, 1971 is not evaluated the same way as 4-14-71. We must often use date functions in order to direct Oracle to use our dates in the manner we wish, a topic covered in a later chapter.

In situations where we wish to return values that are null, we cannot use an equivalence operator. This is due to the fact that a `NULL` has no value; therefore, it cannot be said to be equal to anything. In such situations, we instead use the keyword `IS coupled` with `NULL`. The following screenshot gives us an example of this. We might interpret the statement as, '*Display name and project ID information for employees who have not been assigned to a project*'.



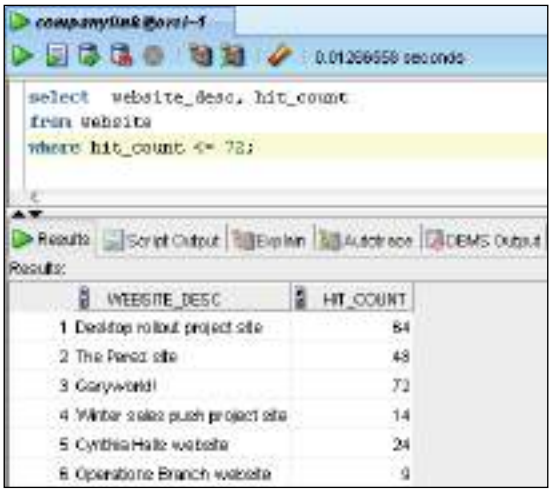
Implementing non-equality conditions

In SQL, it is just as legitimate to query based on conditions that are not equivalent as it is to query based on equality. For such situations, we use conditions of non-equality. The next several examples will use the same examples shown in the section on equality conditions and substitute non-equality conditions instead. The following screenshot displays the first of these:

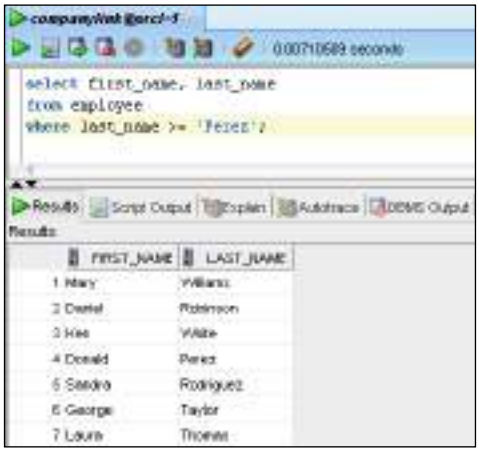


This screenshot introduces the first of our non-equality conditions—'not equal to'. In Oracle's implementation of the SQL language, not equal to is expressed as `<>` (a 'less than' sign paired with a 'greater than' sign). Alternatively, it can be written as `!=`. In this example, we display all rows where the `gender` column does not hold a value of `M`. As before, Oracle scans the table, evaluating the `gender` column; but, in this statement, instead of displaying rows that have a `gender` column value equivalent to `M`, it displays rows that are not equivalent to `M`:

The preceding example makes use of the 'greater than' symbol (>) to evaluate values greater than a given numeric value. This statement could be read as *Display website and hit count information for all sites with more than 72 hits*. Note that, as in mathematics, the use of the 'greater than' sign is non-inclusive; it will not match records equal to 72, only those of greater numeric value. For inclusion, we would need to use the symbol shown in the following screenshot. This example introduces the less than or equal to condition, expressed as <=:



As you can see by the results, this statement is inclusive, returning the rows where `hit_count` is equal to `72`, as well as those of lesser numeric value. While it may seem odd, conditions of non-equality can be used with string literals as well. In the following screenshot, we've requested the first and last name of all employees with a last name that is greater than or equal to the string literal, 'Perez':

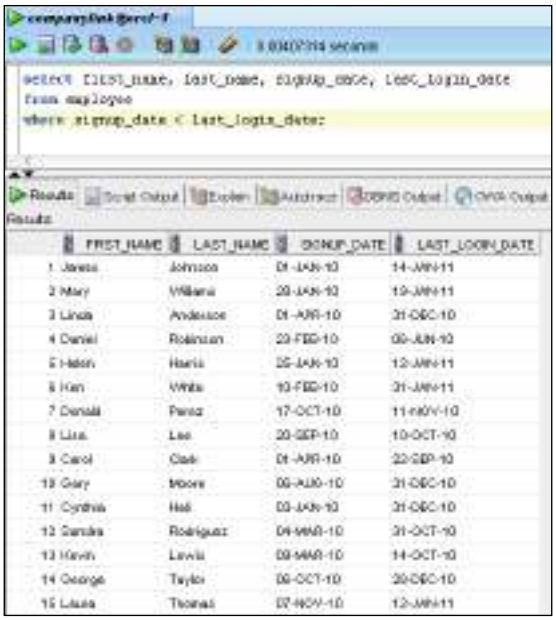


In this circumstance, the query first breaks down the literal Perez into its ASCII values. **American Standard Code for Information Interchange** (**ASCII**) values are the numeric values assigned to all characters so they can be interpreted by computers. Once Oracle has translated the character values into numeric ASCII values, it adds them together. The resulting sum is used as a comparison value from which to determine whether the condition is satisfied.

In the end, any values in the **last_name** column are returned if the last name is greater than the string, 'Perez'. It is important to understand that comparisons such as these are legitimate statements and can be used when the situation calls for it:

This screenshot introduces the idea of using non-equality conditions with date data. Remember that although dates are enclosed in single quotes just as string literals, they are not interpreted as character strings. This example could be read, *Display the employee ID number and date of award for all employees who received an award on or after April 6, 2003*. When we direct Oracle to produce values that are greater than or equal to a given date, the values retrieved will be either the same as the date stored or later than the given date. Thus, when dealing with dates, greater than is interpreted as later than, while less than is earlier than.

When dealing with conditions in a `WHERE` clause, it is often useful to compare the values of two columns instead of comparing a column to a literal value, as we've seen so far. The following screenshot demonstrates this idea:



```
select first_name, last_name, signup_date, last_login_date
from employee
where signup_date < last_login_date;
```

	FIRST_NAME	LAST_NAME	SIGNUP_DATE	LAST_LOGIN_DATE
1	James	Johnson	01-JAN-10	14-JAN-11
2	Mary	Williams	28-JAN-10	19-JAN-11
3	Linda	Anderson	01-APR-10	31-DEC-10
4	Daniel	Robinson	23-FEB-10	09-JUN-10
5	Helen	Harris	25-JAN-10	12-JAN-11
6	Ken	White	10-FEB-10	31-JAN-11
7	Donald	Perez	17-OCT-10	11-NOV-10
8	Lisa	Lee	20-SEP-10	10-OCT-10
9	Carol	Clark	01-APR-10	22-SEP-10
10	Gary	Moore	09-AUG-10	31-DEC-10
11	Cynthia	Hall	03-JAN-10	31-DEC-10
12	Sandra	Rodriguez	04-MAR-10	31-OCT-10
13	Kevin	Lewis	09-MAR-10	14-OCT-10
14	George	Taylor	09-OCT-10	30-DEC-10
15	Laura	Thomas	07-NOV-10	12-JAN-11

In this example, we've implicitly requested that our query scan through the `employee` table and return values where the `signup_date` is earlier than the `last_login_date`. This, of course, returns all the rows in the table, since the fact that an employee has a `last_login_date` implies that they have already signed up for Companylink.

SQL in the real world

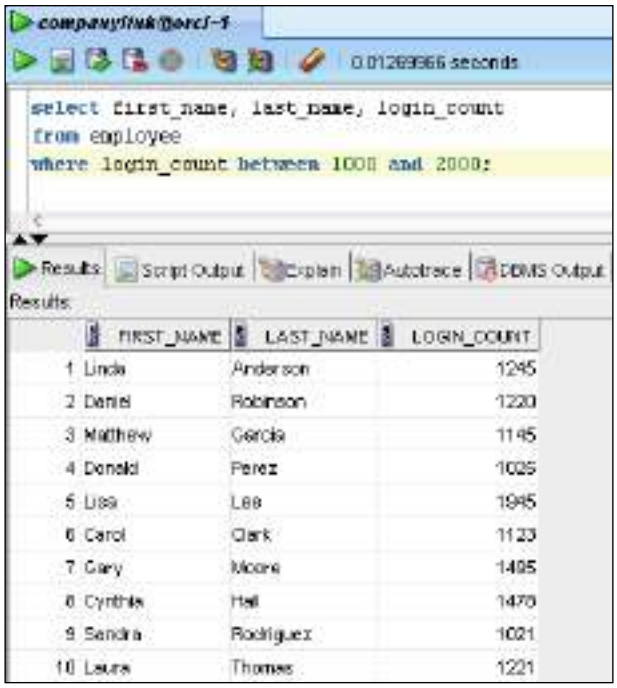
Oracle stores information in date columns not only with date information, but with time as well. It's important to note that, in quality conditions involving dates, two dates are only equivalent if every part of the date, including day, month, year, hour, minute, and second information, is equal. Many date columns are commonly populated with the value for `SYSDATE`, so the time information can be important as well. In situations involving equivalent date conditions that are not returning the rows that you think they should, check to see if the time information may be the reason.

Examining conditions with multiple values

Often in SQL, retrieving the desired data depends more upon matching multiple values than it does on matching a single value. SQL allows us to do this in several different ways. In this section, we examine the use of range conditions, set conditions, pattern-matching, and Boolean operators.

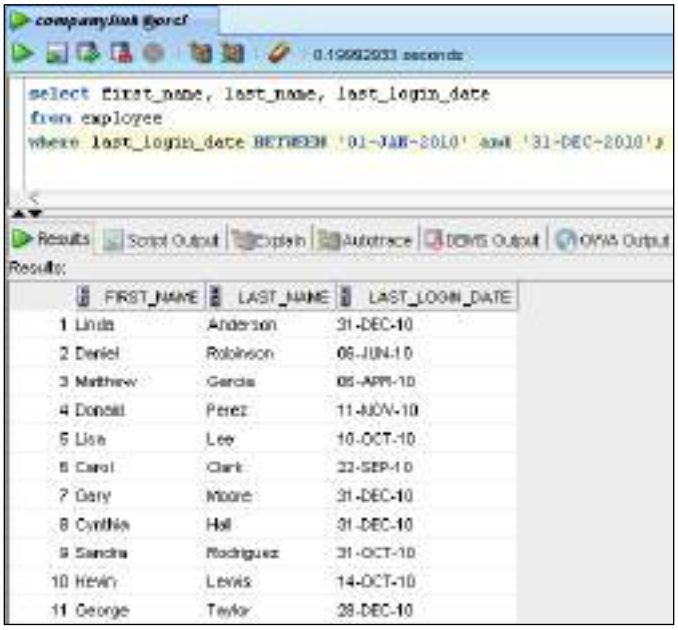
Constructing range conditions using the BETWEEN clause

Say that we want a list of the employees who have logged in to the Companylink site. However, we want to restrict this list to only those who have logged in at least 1,000 times, but no more than 2,000 times. Using what we know of `WHERE` clauses so far, how would we do this? We actually have two conditions that must be met for our result set to meet our requirements. There are multiple ways we could do this, including Boolean conditions, but, for now, we examine the use of a new clause—the `BETWEEN` clause:



While the syntax of the `BETWEEN` clause is fairly natural sounding, it's important to understand several points about how it fits within the overall structure of a conditional SQL statement. First, the `BETWEEN` clause follows the `WHERE` clause and expects two, and only two, conditions. The first condition is the lower bound for the result set, while the second is the upper bound. Second, the `BETWEEN` clause is inclusive; that is, the values for the upper and lower bound are included in the result set. Thus, the statement in the previous screenshot evaluates to greater than or equal to 1,000 and less than or equal to 2,000. Third, the order of the bound values is important. If you were to rewrite the statement as **where login_count between 2000 and 1000;**, no rows would be returned, as there are no values greater than `2000` that are also less than `1000`.

One common use for the `BETWEEN` clause is to specify a date range and to return values that fall within that range. For instance, we want to display name information for employees who last logged into the Companylink site during the calendar year 2010. We could write a SQL statement utilizing the `BETWEEN` clause, as shown in the following screenshot:



The query returns the requested information for `last_login_date` values that fit the given range. No results outside of that range are returned. It is important to note, however, that this query introduces a common mistake when dealing with date ranges. It could potentially exclude some values that should actually be included.

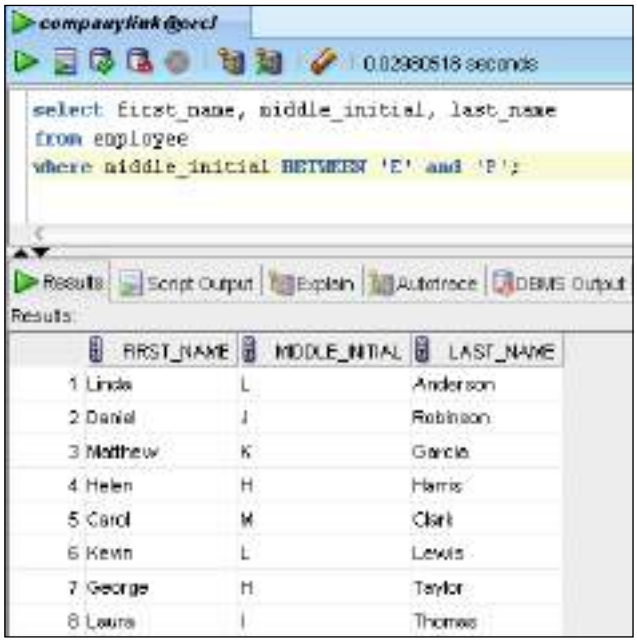
Remember that in Oracle, date columns store not only date information, but time as well. As we have specified no time information in our range, the statement is interpreted as values between 01-JAN-2010 at time 00:00:00 and 31-DEC-2010 at time 00:00:00. Thus, any values after 31-DEC-2010 from time 00:00:00 onwards, such as those inputted at 4:00 PM, would be excluded.

Another possibility would be to write the `where` clause as follows:

```
..where last_login_date between '01-JAN-2010' and '01-JAN-2011';
```

However, this introduces a different problem; namely, that more data might be included than we desire. In this case, any employees that logged in on 01-JAN-2011, at time 00:00:00, would be included. Always be conscious of this issue when dealing with date values.

Although the `between` clause can be used for character values, there are few real-world reasons to do so. As the `BETWEEN` clause only evaluates string literals after breaking them down into their ASCII values, it computes them as essentially a greater than or equal to, less than or equal to. To state that an 'A' is less than a 'G' generally has limited usefulness. However, it can be done, as shown in the following example. Remember, as always, that inside of single quotes, characters are considered case-sensitive:

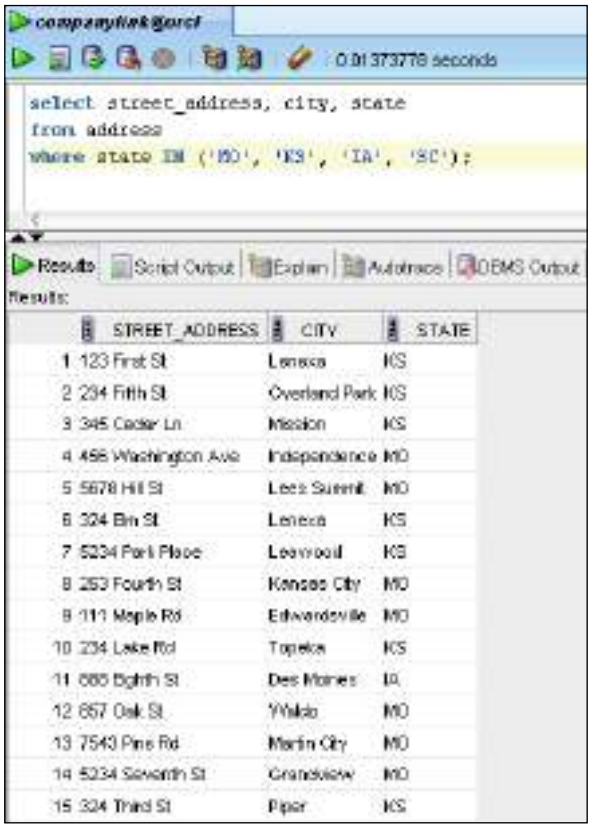


Using the IN clause to create set conditions

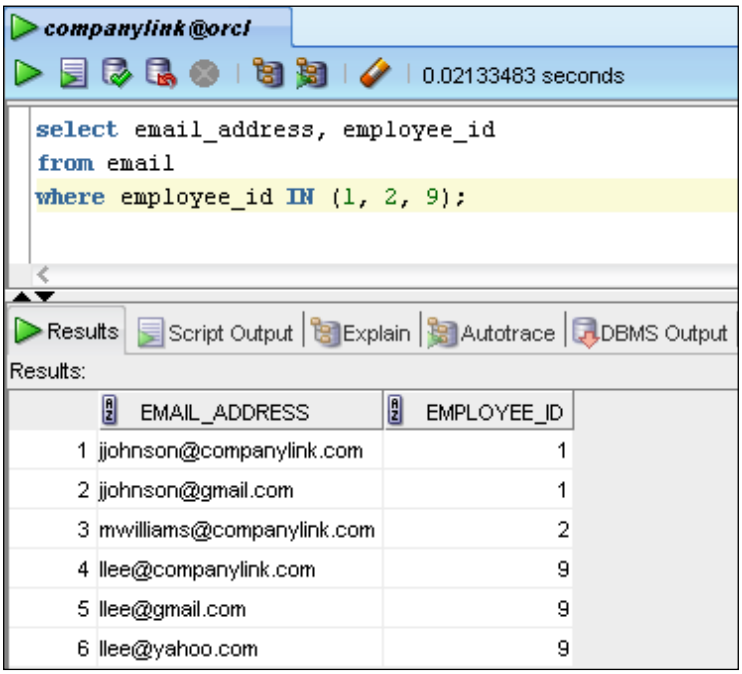
Although range conditions are very useful, it is often just as advantageous to be able to return data based on a more specific set of conditions. Say, for instance, that we want to return data on those employees whose address is in one of four states: MO, KS, IA, or SC. To do this, we could request data where state is equal to MO, KS, IA, or SC, using Boolean values, which is explored later in the chapter. But, evaluating simple lists of conditions in this way is often cumbersome and requires more coding than is necessary. Similarly, we could use a `BETWEEN` with our `WHERE` clause, as in the following code snippet:

```
..BETWEEN 'IA' and 'SC'
```

Unfortunately, that could return more values than we actually require, such as values for OK. A more straightforward way is to use the `IN` clause, which is shown as follows:



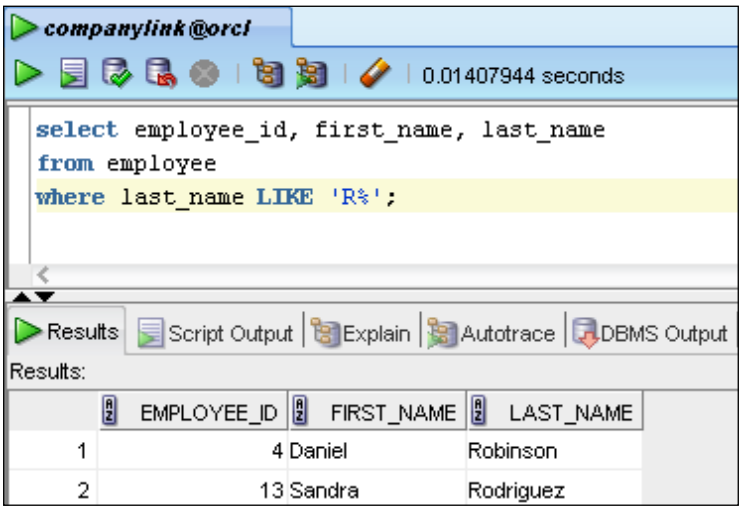
As with the `BETWEEN` clause, we see that the syntax for the `IN` clause is fairly ' natural-sounding'. We request matching data where the row value for state is in a given list of values. These values are listed within parentheses, separated by commas. As always, if the values are string literals, they are enclosed in single quotes. When querying with the `IN` clause, only those rows that match the given list of values will be returned. That fact is borne out in our example, where, as there is no match for the given value 'SC', no data for that value is returned. The `IN` clause can be similarly used with numeric values, as shown in the next example. We could read this as *Show me the e-mail addresses of those employees with an employee ID number of 1, 2, or 9*:



As indicated in the figure caption, this query displays values where the `Employee_id` matches the numeric values `1`, `2`, or `9`. One interesting point to note about this query is that, although there are three matching conditions (1, 2, or 9), more than three values are returned. When, in cases such as `1` and `9`, the `Employee_ID` column has more than one occurrence of a particular value, rows for each of these matching values are returned. For example, there are three occurrences of the value `9` in the `Employee_ID` column in the `email` table. Thus, in our output, three rows are returned that match the value `9`. The `IN` clause can be used with date values as well, although we must take care to list date values that exactly match our requirements. Time information must be taken into consideration as well.

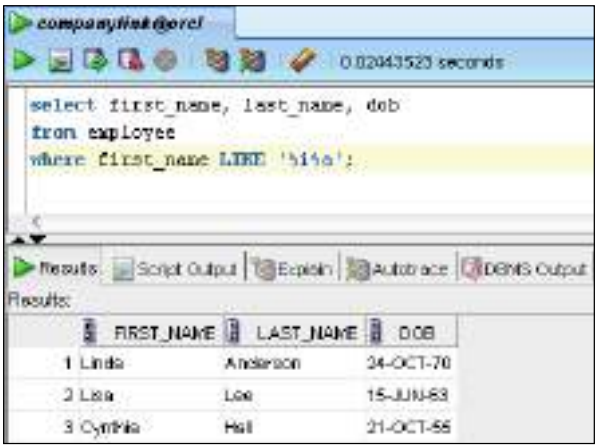
Pattern-matching conditions using the LIKE clause

Sometimes, as with the `IN` clause, we want our row values to match a very specific group of conditions. At other times, it is preferable to match values based on a broader set of conditions. In such situations, we can make use of SQL's powerful pattern-matching capabilities using the `LIKE` clause, as shown in the following screenshot follows. We could interpret this statement as, *Show me the employees whose last names begin with the letter R*.



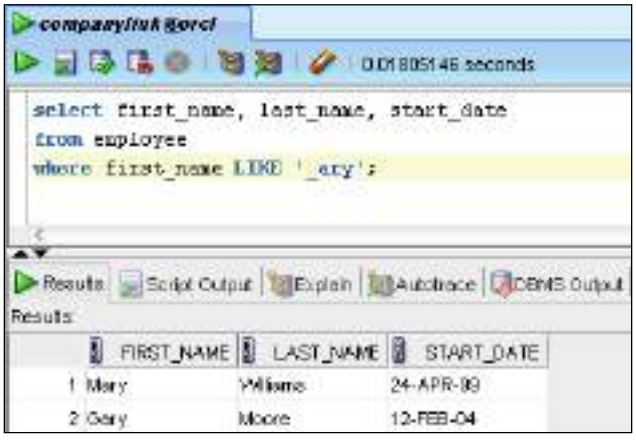
In Oracle's implementation of SQL, pattern-matching is done with two characters: the percent sign (`%`) and the underscore (_), often referred to as **wildcard characters**. The percent sign (`%`) is used to match any number of characters. Thus, our query that requests `LIKE 'R%'` returns any values for `last_name` that begin with an uppercase R and have any number of characters following the R (or no values at all). Thus, the percent sign matches values such as `Robinson` and `Rodriguez`, but would also match string literals such as Raines, Richards, or even just R, if such data existed in our table. The only requirement for matching is that the value begins with an uppercase R. We can also use multiple occurrences of the percent sign for pattern-matching, as shown in the next example.

Here, we pattern-match using multiple percent signs and the `LIKE` keyword:



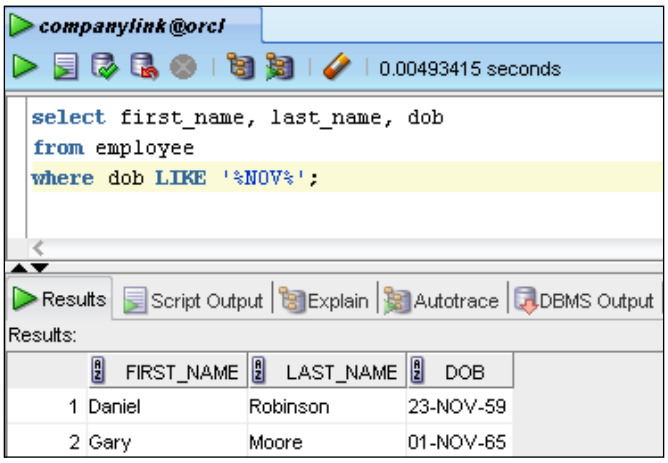
This statement is evaluated as follows. Oracle searches through each value in the `First_Name` column and matches all values that contain any number of characters, then a lowercase i, then any other number of characters, and ends with the letter a. Because the percent sign matches any number of characters, the first occurrence of the letter i doesn't need to be the second character in the name. It could be second—as in the case with `Linda` and `Lisa`, but it also matches `Cynthia`, where the occurrence of the letter i is the sixth character in the name. It is only required that one occurrence of the letter i appears in the name and that the name ends with the letter a.

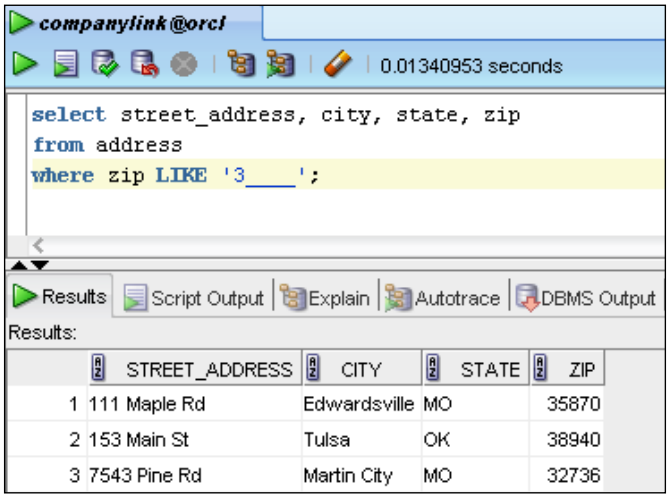
While pattern-matching of any number of values can be done with the percent sign, when we want to use a single character as a wildcard, we use the underscore symbol (_), as shown in the following screenshot:



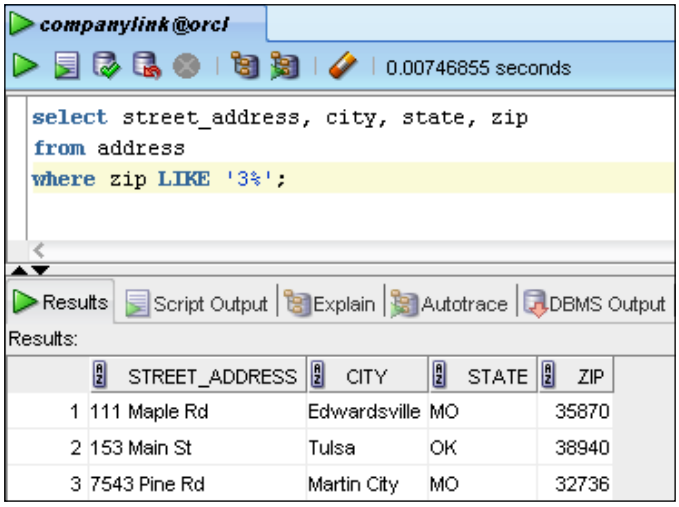
This example matches row values according to this rule: the value must begin with any single character and end with 'ary'. Thus, the names `Mary` and `Gary` are a match, while a name like 'Shary' would not, as there are two characters, 'Sh', before the 'ary' in the name. Both the percent sign and the underscore can be used together in the same condition, if desired.

Although it is much more common to use pattern-matching and `LIKE` with string literals, it can be done with numeric and date values as well, as the following two examples show. The first could be interpreted as, *Display all employees whose birthday is in November*. It uses pattern-matching with a date value. The second could be read as, *Show all addresses whose zip code begins with a number 3*.





The second of the two examples demonstrates the use of multiple underscore characters in pattern-matching. The `LIKE` clause indicates the number 3 followed by four underscores. As all of our zip code values have five digits, we know that a 3 followed by any four numbers will match. The condition shown in the previous example is analogous to the query shown in the following screenshot, which uses a percent sign instead:



For the certification exam, you should be prepared to evaluate complex conditions using both the percent sign and underscore characters.

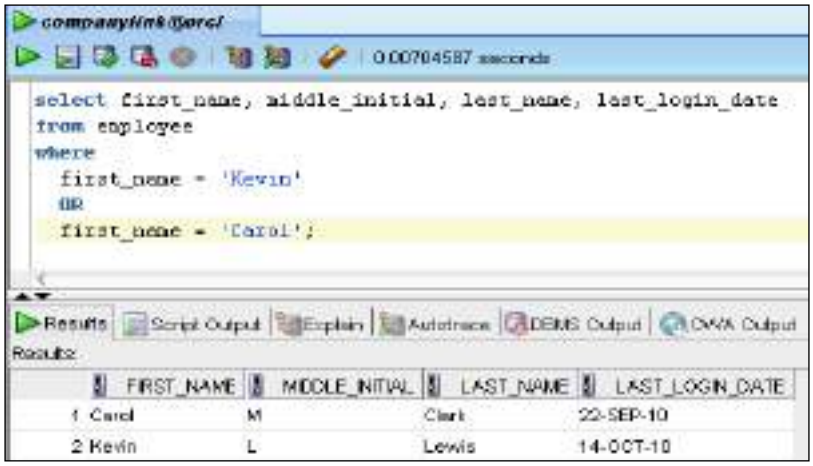
Understanding Boolean conditions in the WHERE clause

The last of the condition types that we will examine are Boolean conditions. In SQL, the Boolean conditions at our disposal are similar to those used in logical mathematics. We use them to attach multiple conditions to a particular query. To accomplish this, we make use of `Boolean` operators, sometimes referred to as logical operators. There are three primary Boolean operators available to us in SQL—the `OR`, `AND`, and `NOT` operators.

Examining the Boolean OR operator

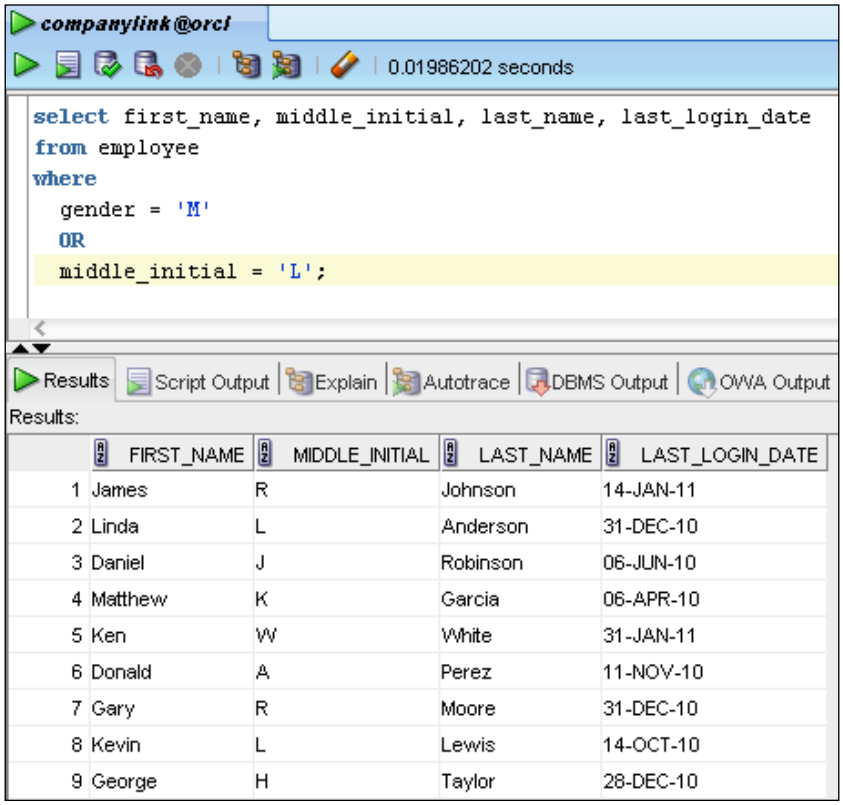
For situations in which we want to return rows that meet any given condition, we make use of the Boolean `OR` operator. As an example, if we attempt to list objects that are either red in color or a fruit, objects such as apples, stop signs, pears, bananas, roses, and fire extinguishers would all suffice, as each of these is either red in color or is a fruit. In a Boolean `OR` statement, at least one of the conditions must be met in order for it to be evaluated as true.

If we apply this idea to the `Companylink` database, we might attempt to write a query that returns all employees whose first name is either 'Kevin' or 'Carol'. The following example lists such a query. It is structured in such a way as to emphasize syntax.



In this example, we return two rows: one where `First_Name` is `Kevin` and another where `First_Name` is `Carol`, as these two rows meet at least one of the conditions.

The real advantage of `Boolean` operators becomes apparent when we use them to concurrently evaluate conditions on different columns, as shown in the next screenshot:



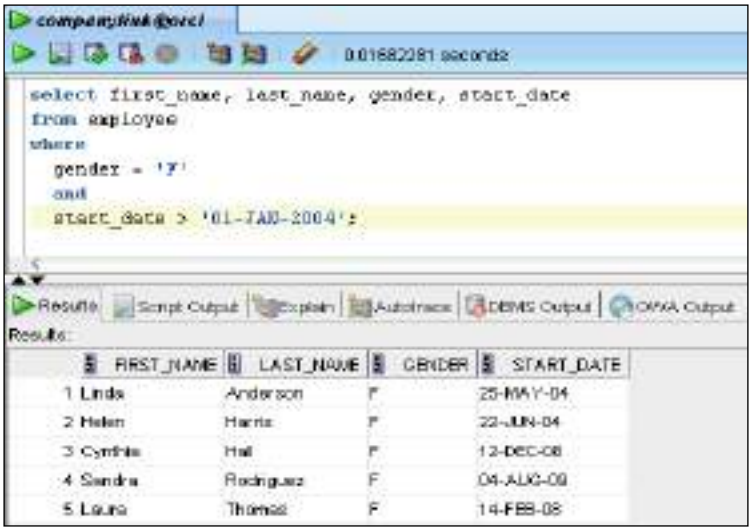
	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	LAST_LOGIN_DATE
1	James	R	Johnson	14-JAN-11
2	Linda	L	Anderson	31-DEC-10
3	Daniel	J	Robinson	06-JUN-10
4	Matthew	K	Garcia	06-APR-10
5	Ken	W	White	31-JAN-11
6	Donald	A	Perez	11-NOV-10
7	Gary	R	Moore	31-DEC-10
8	Kevin	L	Lewis	14-OCT-10
9	George	H	Taylor	28-DEC-10

Here, we evaluate conditions that apply to two different columns—`gender` and `middle_initial`. In order for the row to be returned, it requires that the `gender` column has a value of `M` or that the `middle_initial` column has a value of `L`. If either of these conditions is satisfied, the row will be returned.

Understanding the Boolean AND operator

The Boolean `AND` operator is used in situations where we wish to return rows in which multiple conditions return a value of true. For example, if we wanted to list objects that are both red in color and are also a fruit, an apple would meet both conditions, as an apple is both red and is a fruit. A pear, on the other hand, would not. While a pear would meet one of the conditions, as it is a fruit, it does not meet the condition that the object be red in color. When using the Boolean `AND`, both conditions must be met in order for the statement to evaluate as true.

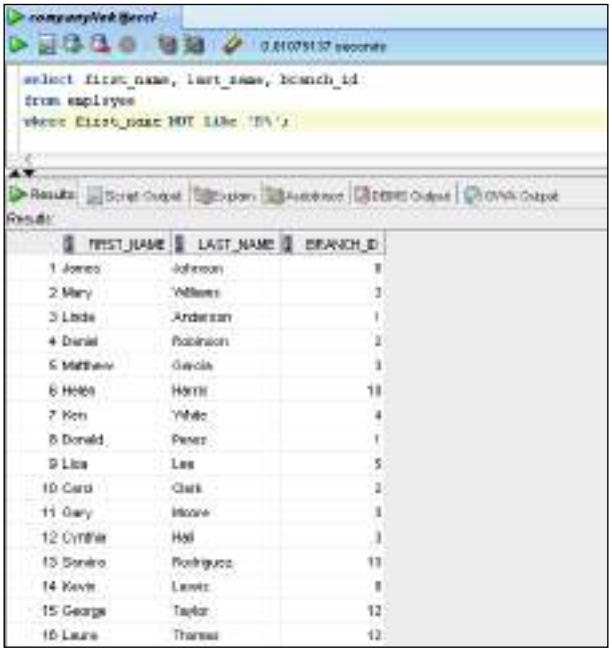
In our `Companylink` database, we might wish to query for female employees that started work after January 1, 2004. A query that satisfies these requirements is shown in the following screenshot:



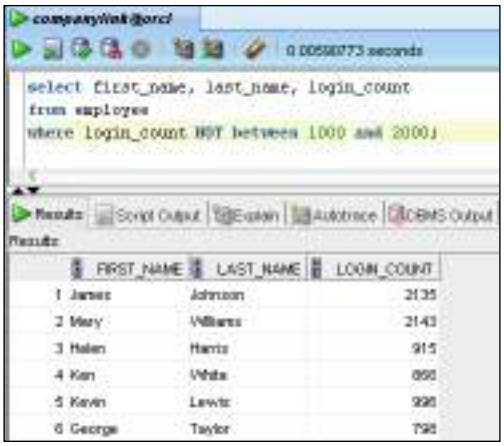
The previous query allows us to use a Boolean `AND` along with conditions for multiple columns. This allows us to form query conditions on two different types of data—`Gender` and `Start_Date`. Only rows that meet both conditions will evaluate as true.

The Boolean NOT operator

Our last Boolean operator is the logical NOT. A `NOT` operator negates the condition that follows it. Thus, the condition itself must evaluate as false in order for the statement to evaluate as true. It's not as confusing as it might sound. An example is shown in the following screenshot:



The easiest way to interpret this query is to first evaluate the statement without the `NOT` operator. As we saw earlier in the chapter, the clause `like 'R%'` will pattern-match all the values in the `First_Name` column that begin with the letter R. When we add the logical `NOT` operator, the opposite is true. The clause `Not like 'R%'` will match all the values that do not begin with the letter R. Thus, the statement is the logical opposite of its predecessor. A `NOT` operator can be used with conditions of equality and the `LIKE`, `BETWEEN`, and `IN` clauses. An example of `NOT` with the `BETWEEN` clause is shown in the following screenshot:



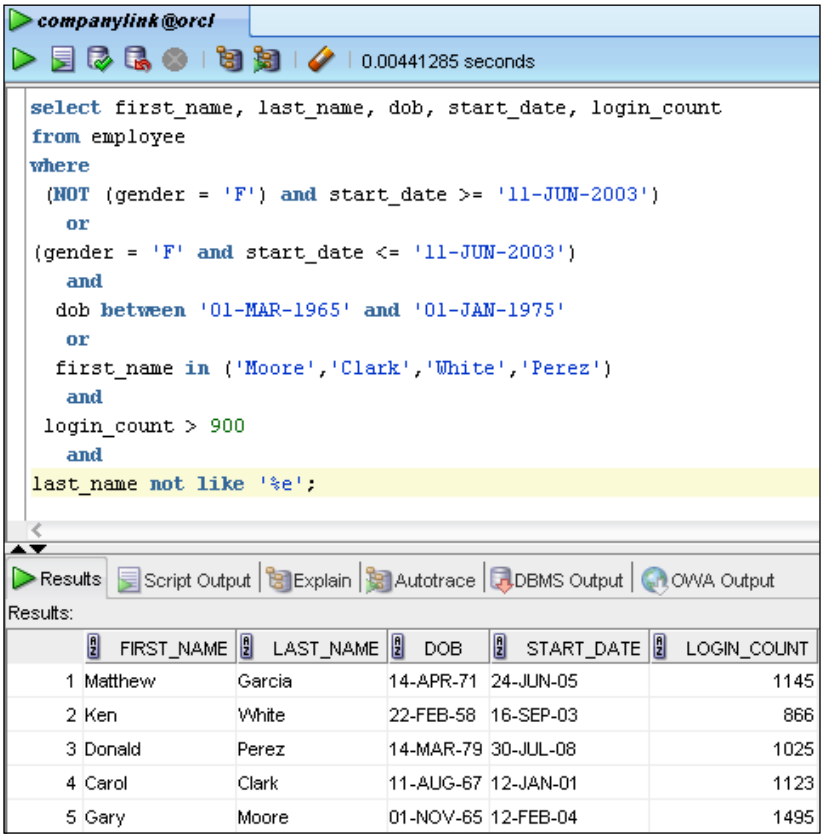
This example is similar to one we used earlier in the chapter, as an example of the `BETWEEN` clause. However, this example utilizes the logical `NOT` operator. As the clause between 1,000 and 2,000 would match values greater than or equal to 1,000, or less than or equal to 2,000, the NOT between 1,000 and 2,000 matches values less than 1,000 or greater than 2,000—all the values not encompassed by the first example.

Boolean operators can be used together or in conjunction with parentheses for grouping, in order to form complex statements with many conditions. In these cases, we must be mindful of the order of precedence in SQL. A short list of this order is as follows:

- Parentheses
- Mathematical operators
- Equality, inequality, and not equal conditions (in that order)
- Boolean operators

Be aware of this order of precedence when examining the next example. It is an advanced statement that makes use of many of the types of conditional operators used in this chapter. It displays employee information for employees that meet several complex conditions, including:

- Female employees with a start date earlier than or including June 11, 2003
- Or male employees who started prior to that date and employees with a date of birth between March 1, 1965 and January 1, 1975, provided that their login count is higher than 900, unless they have a first name listed in the conditions as shown as follows:

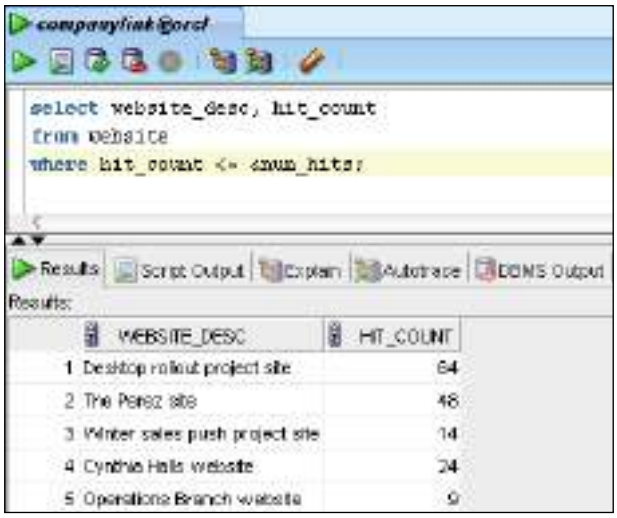


	FIRST_NAME	LAST_NAME	DOB	START_DATE	LOGIN_COUNT
1	Matthew	Garcia	14-APR-71	24-JUN-05	1145
2	Ken	White	22-FEB-58	16-SEP-03	866
3	Donald	Perez	14-MAR-79	30-JUL-08	1025
4	Carol	Clark	11-AUG-67	12-JAN-01	1123
5	Gary	Moore	01-NOV-65	12-FEB-04	1495

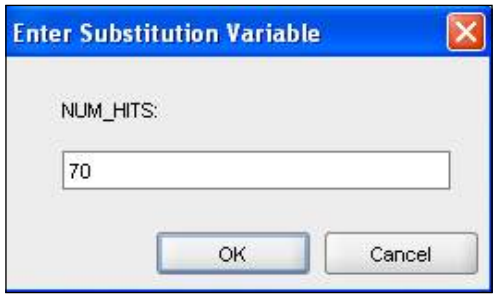
Using ampersand substitution with runtime conditions

In the examples we have used so far, each of the statements had their conditions set before the statement was executed. At the moment of execution, the condition was read and the output determined—no other input was required or allowed from the user. We say that statements such as these have their conditions **hardcoded**—there is no variation in the condition at runtime unless we specifically rewrite it and execute it again. However, in Oracle, we are allowed to vary conditions at runtime using a substitution variable. Runtime substitution variables are denoted in Oracle using the ampersand (&) character.

The following screenshot uses an ampersand variable that prompts us for a value at runtime:



When we run the previous statement, we're prompted with a window that looks like the one shown next:



We can put any numeric value into this prompt that we wish. We are being asked at runtime what numeric condition we want to use, instead of hardcoding it into the statement itself. If, as shown, we enter the number `70` into the prompt, we see the data restricted to the condition less than or equal to 70. If we run the statement again, we are prompted again and can enter a different value, such as `35`, to yield different results, giving us freedom to enter conditional values at runtime as we wish. Note that the prompt we see, **NUM_HITS**, is reflected in the name of the substitution variable, `&num_hits`. Always name your substitution variables in a way that indicates the value for which you are prompting. You can also put more than one ampersand variable into a single query if you need to prompt for multiple conditions. You will be prompted for the first, then any subsequent conditions. If the values are listed in multiple places in the statement and the values you enter are the same, you can use the double ampersand (`&&`) substitution symbol. Additionally, substitution variables can be used for elements other than `where` clause conditions, including the column names for a select statement and table names.

SQL in the real world

While substitution variables might seem convenient, they are not commonly used for the simple reason that they require the user to actually input values, instead of running in batch mode. Substitution variables are more commonly used during development, usually for prototyping, when the ability to test different input values is desired. However, although it is not often used, it is covered on the exam, so it is important that you understand the concept.

Sorting data

One of the most common uses of datasets that have been extracted in the manner we have seen in this chapter is for reporting. Reporting is one of the cornerstones for data usage in today's world, and the ability to sort data in ways that display it for a certain purpose is often crucial. Data that has been extracted by using one or more of our selection methods is useful, but the ability to sort the data adds another dimension to the usefulness of that data.

Understanding the concepts of sorting data

In Oracle, standard tables are referred to as heap-organized tables, which means that the data within them is stored in the order that it was written in. For instance, if I insert data rows into a simple table in the following order—orange, apple, plum—then that is the default order in which our SQL statement will return the data. If we wish to influence this default ordering, we must sort the data. **Sorting** is the ability to take

unordered datasets and display them in a manner consistent with a specified order, such as by date or numeric value. The Oracle architecture is designed to make sorting operations perform at a high level, even utilizing special memory structures to do so. Oracle is capable of doing both numeric and lexicographic (or alphabetic) sorts.

Sorting data using the ORDER BY clause

Take a simple example from our `Companylink` database. We wish to display the list of employees from our `employee` table, but we want to see them displayed in alphabetic order by last name. To do this, we use the `ORDER BY` clause. Its syntax tree is shown as follows:

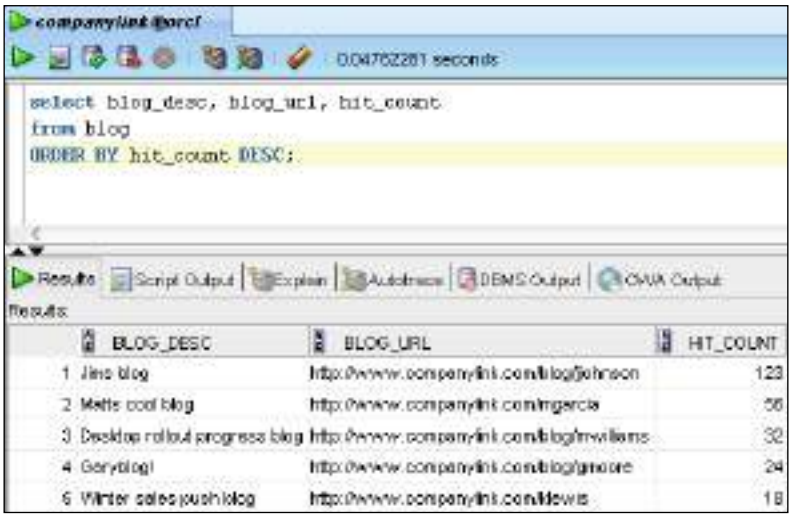
```
SELECT {column, column, ...}
FROM {table}
WHERE {condition}
ORDER BY {column, column, ...};
```

In Oracle, when an `ORDER BY` clause is used, the data is fetched from the database, and then sorted in the requested order, using its internal architecture. Note that the `ORDER BY` clause does not restrict the data returned in any way; it merely displays it in the order requested. The `ORDER BY` clause must also always follow the `FROM` and/or `WHERE` clauses and is always the last clause in a `SELECT` statement. It is also important to understand that our sort conditions are not limited by the columns specified in the `SELECT` clause:

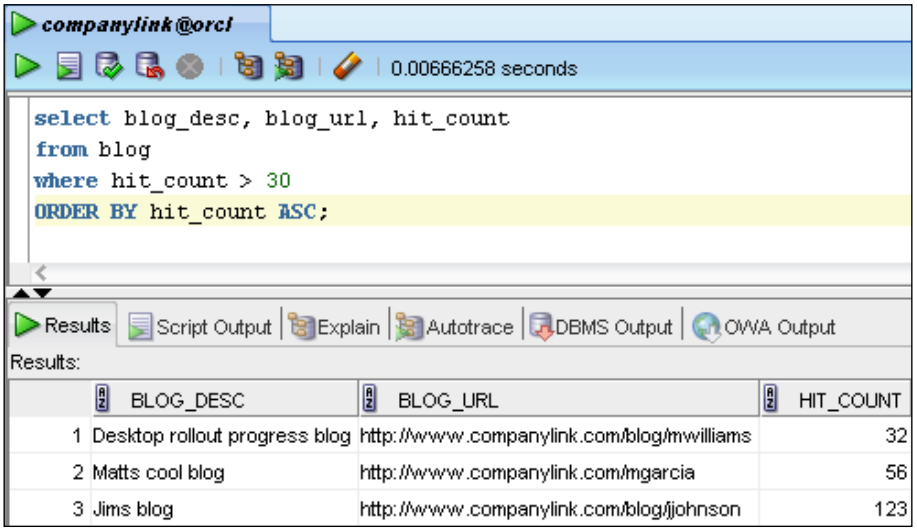
In the previous example, we selected the `Last_Name`, `First_Name`, and `DOB` columns. However, if desired, we could have selected only the `First_Name` and `DOB` columns and still ordered the returned dataset by `Last_Name`. It is not required that we display the column by which we sort, although it is common to do so. As you can see, the requested data is returned and sorted in alphabetical order by the employee's last name. Remember that sorting is not limited to numeric data—it can be used with dates and string literals as well.

Changing sort order using DESC and ASC

When using the `ORDER BY` clause, the default behavior is to sort the data in ascending fashion. However, by adding the `DESC` clause to our `ORDER BY`, we can reverse the sort, as shown in the following screenshot:



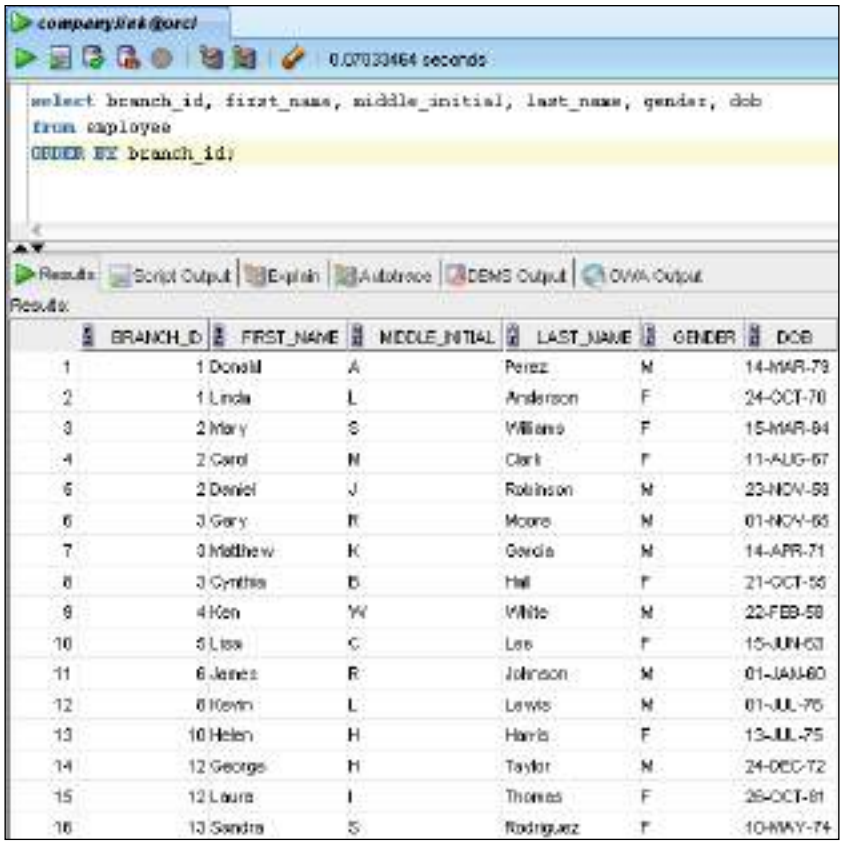
This example demonstrates a descending sort. As we can see, the data is sorted with the largest numeric values at the top and continuing with subsequently lower values. Although ascending sort order is the default behavior, we can still specify it using the `ASC` clause. When we do, the sort behaves just as if we had specified neither `ASC` nor `DESC`. In the next example, we demonstrate this, but we also add a `WHERE` clause to restrict output:



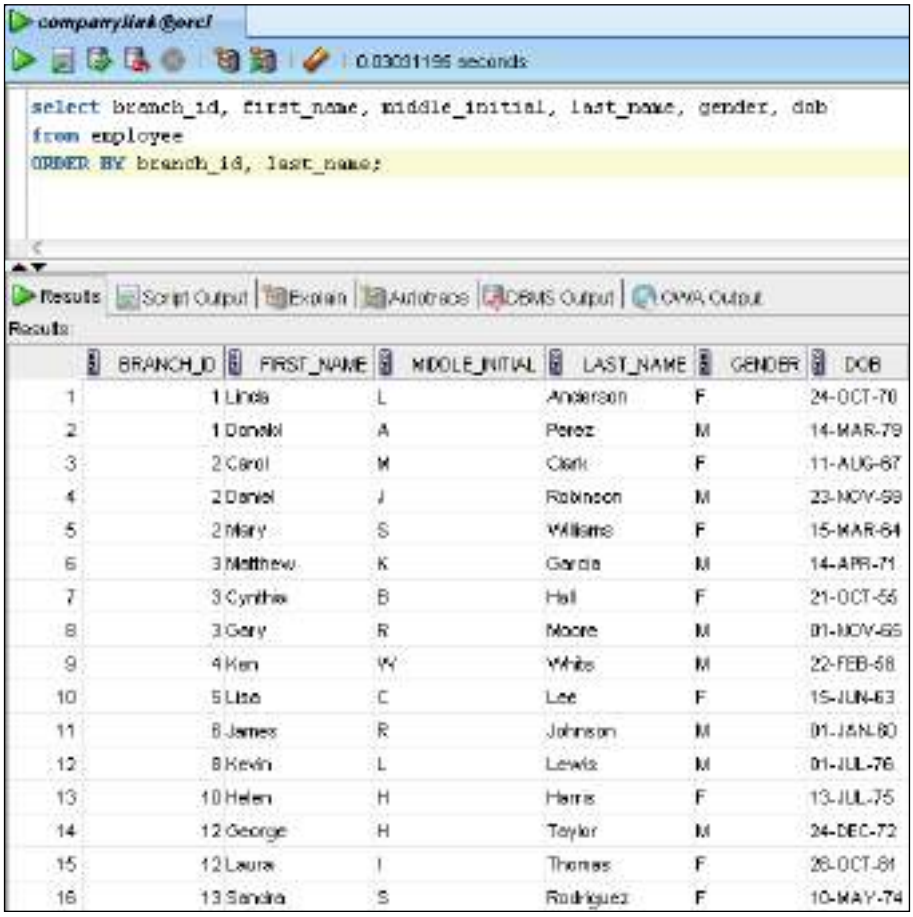
The result returns only three rows, as we have restricted the number of rows to only those matching the condition, greater than `30`. We also display the rows ordered by an ascending value for `Hit_Count`, where lower values are listed first.

Secondary sorts

It is often advantageous to do a 'sort within a sort' when displaying large amounts of data. When we do this, we sort by one value, and then sort any remaining duplicate values by another value. For instance, say that we want to display employee information sorted by the company branch ID to which employees are assigned. To do this, we might use a query like the one listed in the following screenshot:



The results are sorted as we requested, but we see that a number of duplicate values exist for the `Branch_ID` column. Thus, `Mary Williams`, `Carol Clark`, and `Daniel Robinson` have a `Branch_ID` of `2`, but, within that subset of data, their information isn't sorted in any particular order. We may want to sort them again, so that each group of `Branch_ID` has its employees sorted also by their last name. To do so, we need to utilize a **secondary sort**, as shown in the next example:



```
select branch_id, first_name, middle_initial, last_name, gender, dob
from employee
ORDER BY branch_id, last_name;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results

	BRANCH_ID	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	GENDER	DOB
1	1	Linda	L	Anderson	F	24-OCT-70
2	1	Donald	A	Perez	M	14-MAR-79
3	2	Carol	M	Clark	F	11-AUG-67
4	2	Daniel	J	Robinson	M	23-NOV-59
5	2	Mary	S	Williams	F	15-MAR-64
6	3	Matthew	K	Garcia	M	14-APR-71
7	3	Cynthia	B	Hall	F	21-OCT-55
8	3	Gary	R	Moore	M	01-NOV-65
9	4	Ken	W	White	M	22-FEB-68
10	5	Lisa	C	Lee	F	15-JUN-63
11	8	James	R	Johnson	M	01-JAN-80
12	8	Kevin	L	Lewis	M	01-JUL-76
13	10	Helen	H	Harris	F	13-JUL-75
14	12	George	H	Taylor	M	24-DEC-72
15	12	Laura	I	Thomas	F	26-OCT-81
16	13	Sandra	S	Rodriguez	F	10-MAY-74

In this example, the data is essentially sorted twice—once by `Branch_ID`, and then, within each `Branch_ID sort`, by `Last_Name`. This time, those employees with a `Branch_ID` of `2` are shown in the order `Clark`, `Robinson`, `Williams`, which is the alphabetical order for `Last_Name`. When doing secondary sorts, this statement can be read as, *Sort by Branch_ID, then sort within Branch_ID by Last_Name*. The number of secondary (or tertiary) sorts we can do in this manner is only limited by the number of columns that exist in the table. In the following example, we sort by three columns. It could be expressed as, *Display employee information sorted by the employee's branch ID, then by their gender, then by their last name*.

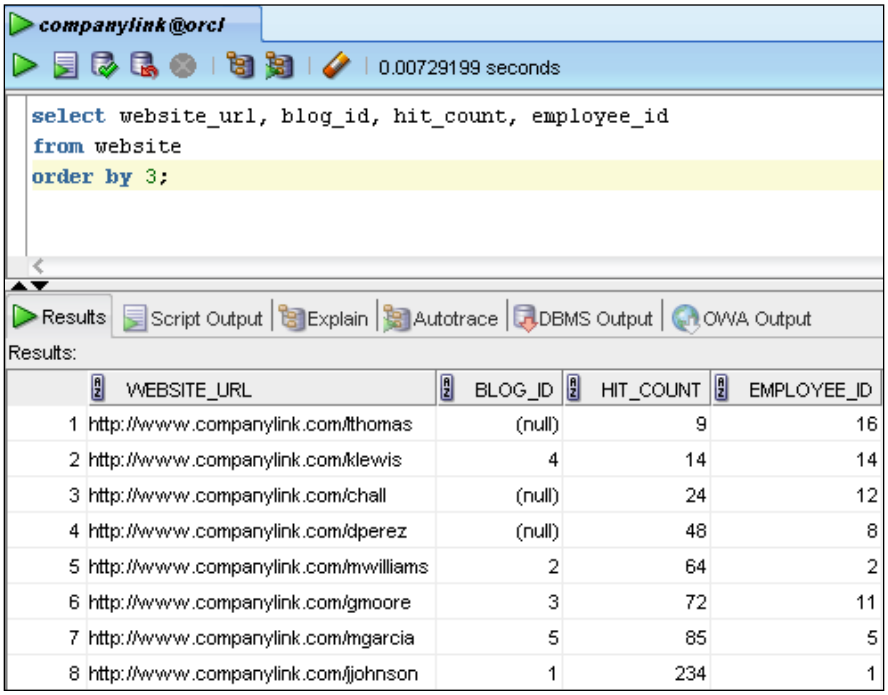
```
select branch_id, first_name, middle_initial, last_name, gender, dob
from employee
ORDER BY branch_id, gender, last_name;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	BRANCH_ID	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	GENDER	DOB
1	1	Linda	L	Anderson	F	24-OCT-70
2	1	Donald	A	Perez	M	14-MAR-79
3	2	Carol	M	Clark	F	11-AUG-67
4	2	Mary	S	Williams	F	15-MAR-64
5	2	Daniel	J	Robinson	M	23-NOV-59
6	3	Cynthia	B	Hall	F	21-OCT-55
7	3	Matthew	K	Garcia	M	14-APR-71
8	3	Gary	R	Moore	M	01-NOV-65
9	4	Ken	W	White	M	22-FEB-58
10	5	Lisa	C	Lee	F	15-JUN-63
11	6	James	R	Johnson	M	01-JAN-60
12	8	Kevin	L	Lewis	M	01-JUL-78
13	10	Helen	H	Harris	F	13-JUL-75
14	12	Laura	I	Thomas	F	26-OCT-81
15	12	George	H	Taylor	M	24-DEC-72
16	13	Sandra	S	Rodriguez	F	10-MAY-74

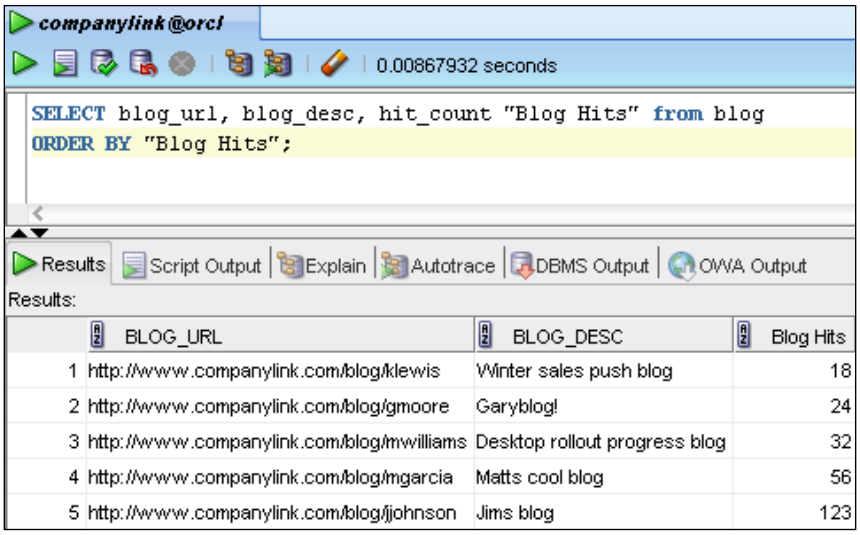
There is one additional type of sort that we can do in Oracle—the **positional** sort. The positional sort allows you to sort by the positional number of the column, not the column name itself. This type of sort is shown in the following screenshot:



	WEBSITE_URL	BLOG_ID	HIT_COUNT	EMPLOYEE_ID
1	http://www.companylink.com/tthomas	(null)	9	16
2	http://www.companylink.com/klewis	4	14	14
3	http://www.companylink.com/chall	(null)	24	12
4	http://www.companylink.com/dperez	(null)	48	8
5	http://www.companylink.com/mwilliams	2	64	2
6	http://www.companylink.com/gmoore	3	72	11
7	http://www.companylink.com/mgarcia	5	85	5
8	http://www.companylink.com/jjohnson	1	234	1

The data is displayed, sorted by `Hit_Count` (the third column listed in the select statement). Notice that `Hit_Count` is not the third column in the table (it is the fifth), but it is the third column listed in the `SELECT` statement. So, to say, *Sort data by the third column in the dataset* is an accurate representation of what is occurring, whereas, *Sort data by the third column in the table* is not.

Also, notice that we can sort using not only a column name, but a column alias as well. In these situations, we simply define the alias in the `SELECT` clause and then reference it in the `ORDER BY` clause. The following example shows the preceding query rewritten to sort by the alias '`Blog Hits`':



Summary

In this chapter, we've added two major skills to our growing list of SQL capabilities: selectivity and the ability to sort data. We've examined the structure of both the `WHERE` clause and the `ORDER BY` clause. We've looked at a large number of condition types that can be used with the `WHERE` clause, including equality, non-equality, and Boolean conditions. We've explored conditions that require additional clauses, such as range, set, and list conditions. We've examined how Oracle utilizes pattern-matching and substitution variables to create open-ended conditions. Finally, we've worked with different ways to sort our resulting sets of data.

Certification objectives covered

In this section, we have seen the following certification objectives covered:

- Limiting the rows that are retrieved by a query
- Sorting the rows that are retrieved by a query
- Using ampersand substitution to restrict and sort output at runtime

Up to this point, we've focused almost exclusively on ways to retrieve data from database tables. In the next chapter, we'll add a completely new dimension to our SQL abilities—the ability to manipulate the data in existing tables. We'll learn how to add rows to a table, to change data in tables, and to delete data from tables.

Test Your Knowledge

1. Which of these is a term used to describe the ability to limit data sets to rows that meet certain conditions?
   a. Projection
   b. Restriction
   c. Selection
   d. Registration
2. Which of these statements represents the proper use of the SQL WHERE clause syntax?
   a. select first_name, last_name where last_name = 'Johnson';
   b. select first_name, last_name from employee;
   c. select first_name, last_name where last_name = 'Johnson' from employee;
   d. select first_name, last_name from employee where last_name = 'Johnson';
3. Which of these is NOT a type of condition that can be used in SQL?
   a. Boolean
   b. Equality
   c. Non-equality
   d. Ascending
4. Which of these is NOT a symbol that can be used in non-equality conditions?
   a. >
   b. <=
   c. <>
   d. ||

5. Which of these statements would correctly satisfy the request, Display the message text where the message ID is greater than or equal to 4 and less than or equal to 8?
   a. Select message_text from message
      where message_id between 4 and 8;
   b. select message_text from message
      where message_id >= 4 and <= 8;
   c. select message_text from message
      where message_id > 4 and message_text < 8;
   d. select message_text from message
      where message_id between 8 and 4;

6. Which of the following statements is a correct use of set conditions using the IN clause?
   a. Select first_name, last_name from employee
      where employee_id in 1, 5, 12;
   b. select first_name, last_name from employee
      where employee_id in (1, 5, 12);
   c. select first_name, last_name from employee
      where employee_id = (1, 5, 12);
   d. select first_name, last_name from employee
      where employee_id in (1 5 12);

7. Which of the following statements would satisfy the requirement, Display employee data for all employees whose first name begins with the letter R?
   a. select first_name, last_name from employee where last_name like 'R%';
   b. select first_name, last_name from employee where first_name like '%R';
   c. select first_name, last_name from employee where first_name like 'R%';
   d. select first_name, last_name from employee where last_name like '%_R_';

8. From your Companylink database, which of the following statements would return the values 'Johnson', 'Anderson', and 'Robinson' when executed?
   a. select last_name from employee where last_name like '__son';
   b. select last_name from employee where last_name like 'R%son';
   c. select last_name from employee where last_name like '%son_';
   d. select last_name from employee where last_name like '_%son%';
9. Which of the following is NOT a legal Boolean logical operator?
   a. AND
   b. IN
   c. OR
   d. NOT
10. Which of the following correctly uses a Boolean operator as a condition?
    a. select first_name from employee
       where last_name = 'White' or last_name = 'Lee';
    b. select first_name from employee
       where first_name = 'Mary' or 'Ken';
    c. select first_name, gender from employee
       where gender = 'F';
    d. select first_name, dob from employee
       where first_name is not 'Carol';
11. Which of the following statements is equivalent to the statement?
    select * from blog where hit_count between 40 and 80;'?
    a. select * from blog where hit_count in (40, 80);
    b. select * from blog where hit_count >= 40 or hit_count <= 80;
    c. select * from blog where hit_count like '40%80';
    d. select * from blog where hit_count >= 40 and hit_count <= 80;

12. Which of these is a proper substitution variable in SQL?
    a. *new_employees
    b. %num_rows
    c. ||highest_hit_count
    d. &choose_your_table
13. Which of these statements about sorting is NOT true?
    a. Oracle is capable of doing both numeric and lexicographic sorts
    b. Sorting is achieved through use of the ORDER BY clause
    c. Data is automatically sorted when it is inserted into Oracle tables
    d. The ORDER BY clause always follows the FROM clause in SQL
14. Which of these statements will correctly sort data from high to low?
    a. select * from blog order by hit_count;
    b. select * from website order by hit_count DESC;
    c. select * from blog order by hit_count ASC;
    d. select * order by hit_count from website;
15. Which of the following statements will correctly sort data, first by the employee's gender, then by their last name?
    a. select first_name, last_name, gender from employee
       order by last_name, gender;
    b. select first_name, last_name, gender from employee
       order by gender, first_name;
    c. select first_name, last_name, gender from employee
       order by 2, 1, 3;
    d. select first_name, last_name, gender from employee
       order by gender, 2;

4

Data Manipulation with DML

Thus far, our efforts to utilize database data have been limited to *pulling* data from the database. We've looked at how to select data, how to limit it based on conditions, and how to sort it. However, we've only worked with data that already existed in our `Companylink` database, which was created back in *Chapter 1*, *SQL and Relational Databases* when you ran the `companylink_db.cmd` batch file. In this chapter, we examine some of the types of statements that were actually used in that script. These commands allow us to create and modify data in our Companylink database.

In this chapter, we shall:

- Examine the concept of DML—Data Manipulation Language
- Create data using the `INSERT` statement
- Copy data using `INSERT/SELECT`
- Modify data using the `UPDATE` statement
- Remove data using the `DELETE` statement
- Examine transaction control
- Learn to recognize and debug errors

Persistent storage and the CRUD model

The ability to manipulate data within an RDBMS requires an understanding of two primary aspects—the concept of persistent storage and the syntactical rules for manipulation. In this section, we examine these concepts.

Understanding the principles of persistent storage

The basic principles of computing dictate that, under normal conditions, data is operated on within RAM, or Random Access Memory. RAM has two important characteristics: it is fast, and it is volatile. Compared to the access speeds of hard disk, RAM can access data an order of magnitude faster. This makes it ideal for the computation and manipulation of data. Processes can operate quickly and efficiently to read and change data within RAM. However, the integrated circuits that make up RAM also make it volatile, in the sense that data stored within it cannot survive the loss of power to the computer system. If we turn off the power, the data in RAM is lost. The volatility of RAM is what necessitates the need for hard disk storage. A hard disk, or hard drive, is capable of **persistent storage**—data storage that can outlive a system power loss.

An RDBMS such as Oracle makes use of this architecture to its fullest. In Oracle, the vast majority of data manipulation takes place in memory structures in RAM called caches. Data is cached in shared memory in order to make it available to many processes simultaneously. At various intervals, the data in cache is written out to disk, and the data is stored persistently. This makes the RDBMS resistant to data loss from power outages and contributes to a more stable system.

This is the underlying architecture at work. For our purposes, we are concerned with how we can utilize the SQL language to add, modify, and remove data from this persistent storage layer. To do so, we use DML—Data Manipulation Language.

SQL in the real world

Although the methods of persistent storage listed previously are still the standard way a computer operates, as of this writing, there are many advances underway that may make them a thing of the past. Research into topics such as non-volatile RAM, RAM that can store data beyond a power loss, and Solid State Disks (SSD), hard disks that can retrieve data at speeds approaching that of RAM, may soon change how computers operate at a very fundamental level.

Understanding the CRUD model and DML

Historically, persistent storage in databases has been managed according to a generic standard known as the **CRUD model**: **C**reate, **R**ead, **U**pdate, **D**elete. These four operations form the fundamental ways that data is utilized in a database. Data can be created, or placed into the database; it can be read from the database; it can be changed, or updated; and it can be deleted from the database. Some define the CRUD model as Create, Retrieve, Update, Destroy, but the individual functions themselves are the same. These four operations denote the way that users interface with database data.

In SQL, the CRUD model is implemented as **DML**, or **Data Manipulation Language**. DML is generally considered a **sublanguage** or subset of the SQL language as a whole. As we will see throughout this book, there are many types of SQL statements. The commands that make up DML are a part, or subset, of the entire language. DML defines persistent storage operations through the use of four commands: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, sometimes referred to with the acronym SIUD. Although the terms are different than those used in the CRUD model, the four functions are the same. A side-by-side comparison is shown in the following chart:

CRUD operation	DML command	Function
Create	Insert	Creating or adding new data into the database
Read/Retrieve	Select	Retrieval of data from the database
Update	Update	Modifying existing data within the database
Delete/Destroy	Delete	Removing existing data from the database

These operations enable the use of **data manipulation** in an RDBMS. Through DML commands, we can change the state of data that is in persistent storage.

SQL in the real world

Considering that they don't actually change data in persistent storage, you might be wondering why `SELECT` statements belong as a part of DML. In fact, many believe they don't. The entire subject of SQL sublanguages is fairly broadly defined. Some consider SQL itself to be a sublanguage. Others believe `SELECT` statements are a part of their own sublanguage. Still others claim that `SELECT` statements are a part of DCL, or Data Control Language. It's not that important to get caught up in the terminology debate. General concensus puts the `INSERT`, `UPDATE`, and `DELETE` statements squarely in the category of DML.

Creating data with INSERT

Our first look at manipulating data involves one of the most fundamental operations in an RDBMS—the ability to add data. This section looks at ways to create data within an Oracle database.

Examining the syntax of the INSERT statement

The primary command in SQL that allows users to create the data that resides in tables is the `INSERT` statement. With it, we can add rows to an existing table, provided that the format we use for the statement fits with the existing column structure. The syntax tree for the basic `INSERT` statement is as follows:

```
INSERT INTO {table_name}
VALUES (value1, value2, ... );
```

Considering the lengths to which we've taken our `SELECT` statements thus far, it may seem that the syntax for the `INSERT` statement is considerably more straightforward. Generally, this is the case, although the degree of simplicity in an `INSERT` is dependent on the simplicity of the table itself. If we examine our basic `INSERT` statement, the first keyword we encounter is `INSERT`, followed by `INTO`. `INSERT INTO` is followed by a table name that we specify. This will be the table to which we want to add data. The last clause in our syntax tree is the `VALUES` clause, which specifies the actual row data we intend to add. This list of values is enclosed in parentheses. We will examine different ways that values can be specified, but it is important to remember that the data we indicate must match the table structure itself, in terms of both column structure and datatype.

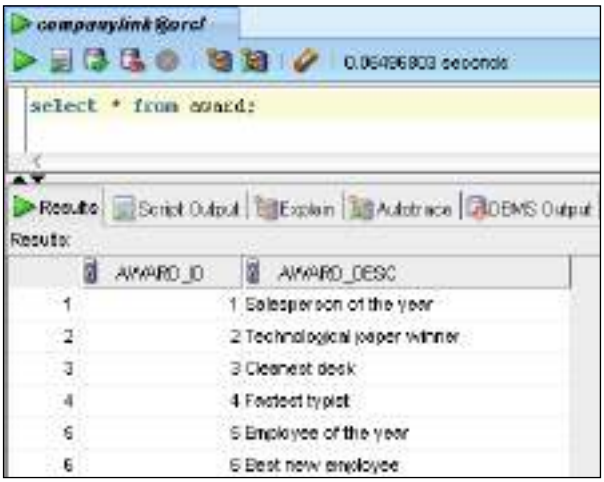
When we specify the list of values in the `VALUES` clause as shown in the next screenshot, it is implied that we are *listing them in the order that they appear in the table*. Say, for example, that we have a table with two columns: `object` and `color`. If we insert two values into our table, `apple` and `red`, it is implied that we are inserting the value `apple` into the first column, `object`, and the value `red` into the second column, `color`. We refer to this method as **positional**—the order that the data is inserted is the same as the position of the columns in the table. The first value goes into the first column; the second value into the second. The list of values must also match in terms of datatype. If we attempt to insert numeric data into a column that holds character string data, we will receive an error. Thus, in our example, if we assume that our columns `object` and `color` are character data, our list of values cannot contain numbers or dates.

Using single table inserts

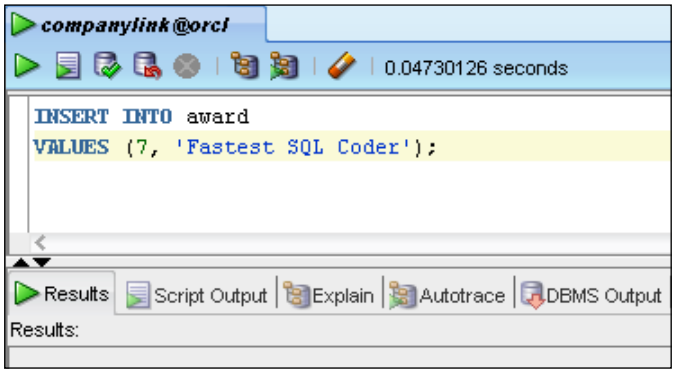
The most fundamental `INSERT` statements add rows to a single table one row at a time. To accomplish such operations, we can use either of the two primary syntactical notations—positional or named column.

Inserts using positional notation

Let's take what we've learned so far and add data to our `Companylink` database; specifically, the `award` table. The following query shows the table before our `insert` as a point of reference:

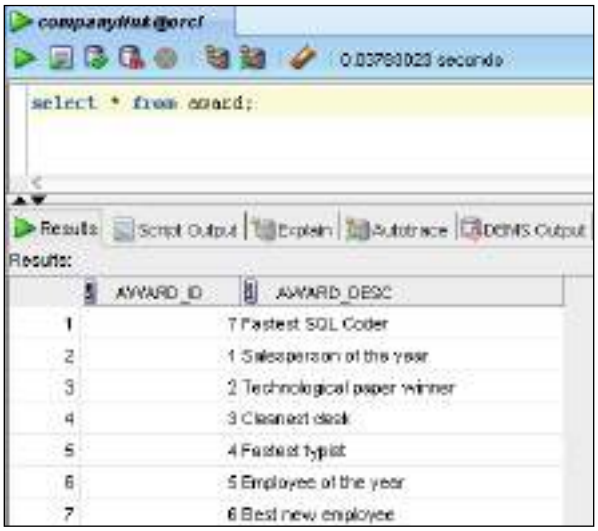


As we can see, the `award` table contains two columns, `award_id` and `award_desc`, and six rows. Following our `INSERT` statement, we should see one additional row with the data that we specify. The example of the `INSERT` statement is shown as follows:



Here, we invoke our `INSERT` statement and specify the `award` table as our target for data insertion. Taking the columns in order, we specify the numeric value **7** and the character string value **'Fastest SQL Coder'** to insert into the `award_id` and `award_desc` columns, respectively. Again, the values are enclosed in parentheses. We see the results in the following screenshot. Notice that even though our row was added after the existing six rows, the row we've inserted displays as the *top* row in the table. This behavior is unique to SQL Developer and has to do with how the tool processes matrices of data. You may not see this behavior in other tools.

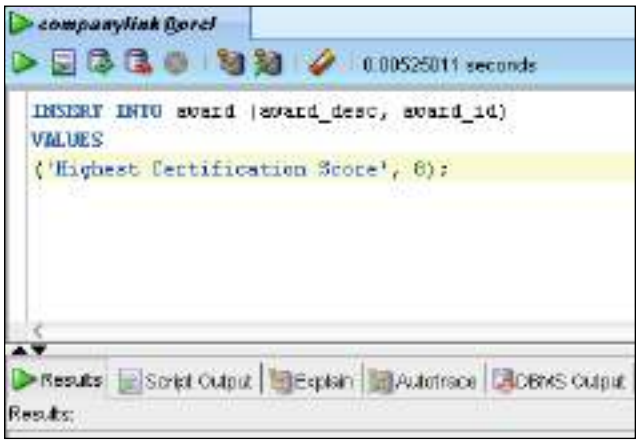
Even though it looks unusual, there is nothing actually wrong with the data being displayed in this way. Oracle tables are classified as heap-organized, which means they are organized in no particular order with respect to row. Always remember to use an ORDER BY clause if the order of the data is important.



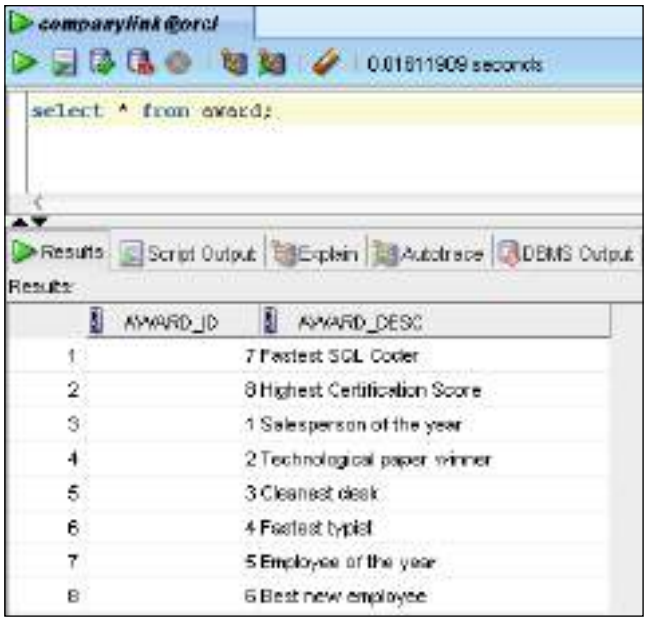
Our results now show seven rows where there were six before. We specified our values in the same order as that of the columns, and we see them displayed as such.

Inserts using named column notation

It is not necessary to let the column order dictate the order of our values. Often it is desirable to specify values in an order that we choose. For this, we use **named column notation**. We specify a list of columns in the same order as that of our values. An example using named column notation and its results are shown in the following two screenshots. First, we run the `INSERT` statement itself.



Next, we view our results using a simple `SELECT` statement.



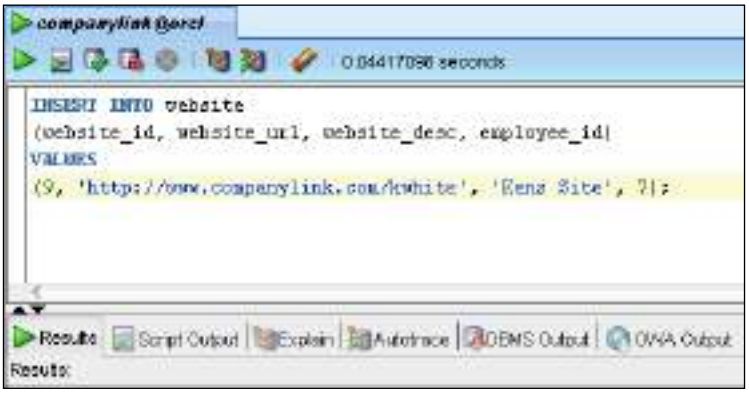
In the first of these two statements, we notice two differences from our previous `INSERT` statement. First, following the table name we've specified, `award`, we see a list of column names enclosed in parentheses. These are the two columns in the `award` table, but we've specified them in a different order than they occur in the `award` table. Because we do this, we can also specify our values in an order different from that of the table. The first value in the list, **Highest Certification Score**, is inserted into the first column specified, **award_desc**, and the second value, **8**, is inserted into the second column specified, **award_id**. We can do this with any number of columns and values, provided that all the columns exist in the table and that their datatypes match. If we wish to designate our columns this way, we can order them any way we wish, even if that order is the same order of the columns in the table.

SQL in the real world

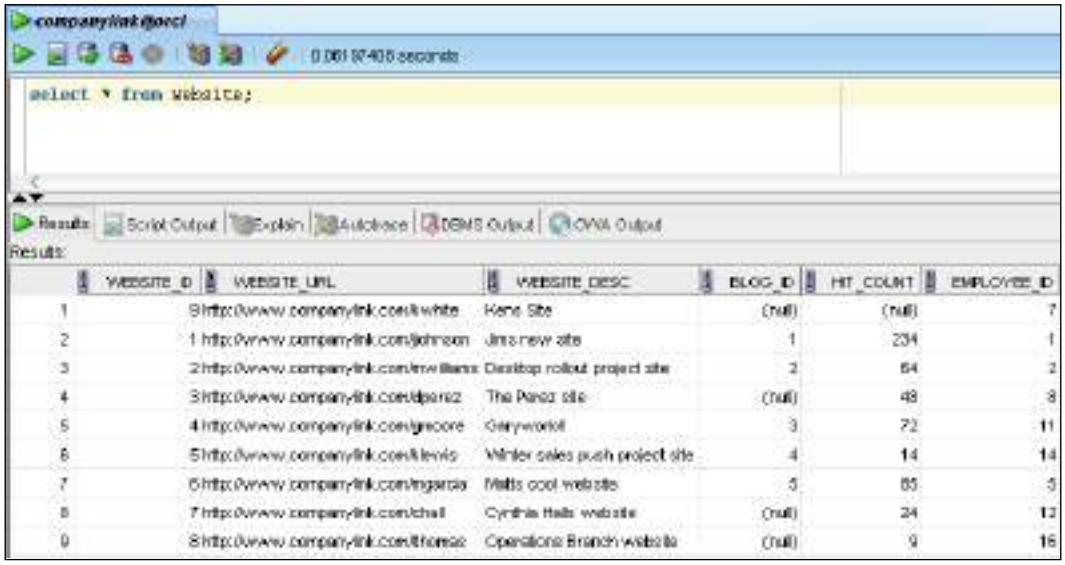
In the development world, you will often find that the coding standards of a particular company prohibit the use of positional notation, instead favoring named column notation. This is a responsible practice, since using named column notation is much more readable. With named column notion, it is much simpler to determine the values that belong with certain columns.

Inserts using NULL values

As we've discussed before, it is possible to have rows where a particular column has no value. We refer to this lack of value as a NULL value. We can enter null values into a table in several ways. The first of these is to use named column notation to specify values for a number of columns that is fewer than actually exist in the table. When we do this, the remaining columns will be populated with nulls. We see this in the following example:



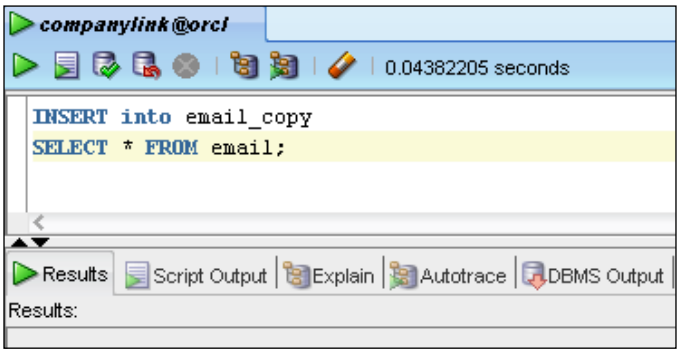
There are six columns in the `website` table. But, we see from this example that we are only specifying four columns and four values in our `insert`. In this statement, the values will be inserted according to the order determined by our named column list. The remaining two columns, `blog_id` and `hit_count`, will be populated with nulls. We see the results in the next query. Notice the row where **website_id** equals **9**—in it; `blog_id` and `hit_count` are null.



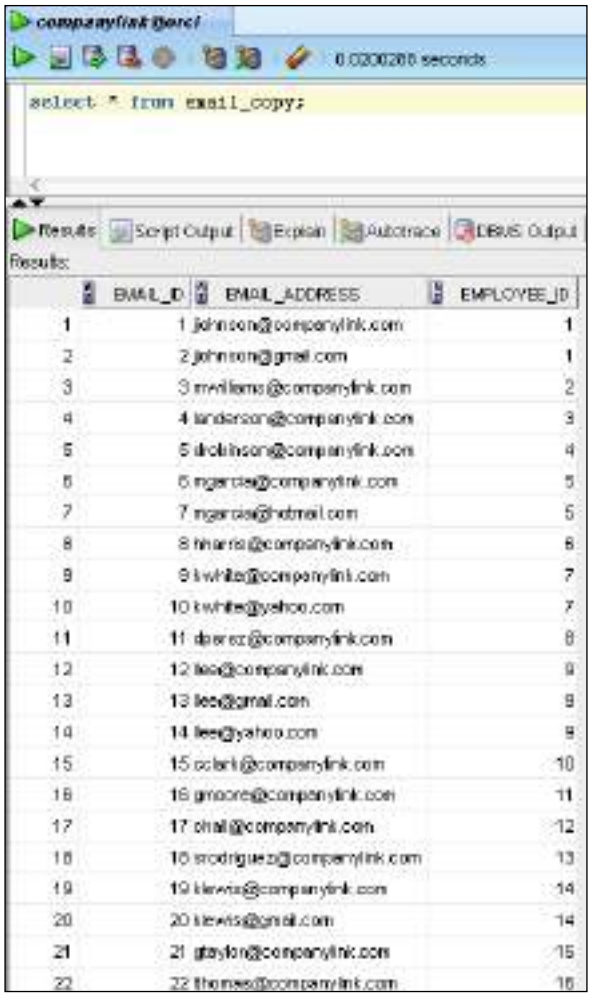
Not all columns can contain nulls. In later chapters, we will see how to construct rules that forbid certain actions from being taken on our tables, including the insertion of null values. These rules are called constraints.

Multi-row inserts

Our previous `INSERT` statements have resulted in the addition of one row per statement. As we stated, these were single row inserts. With these types of statements, if we want to insert multiple rows, we must run multiple `insert` statements. However, we can also insert multiple rows into a table using multi-row inserts as shown in the following screenshot:



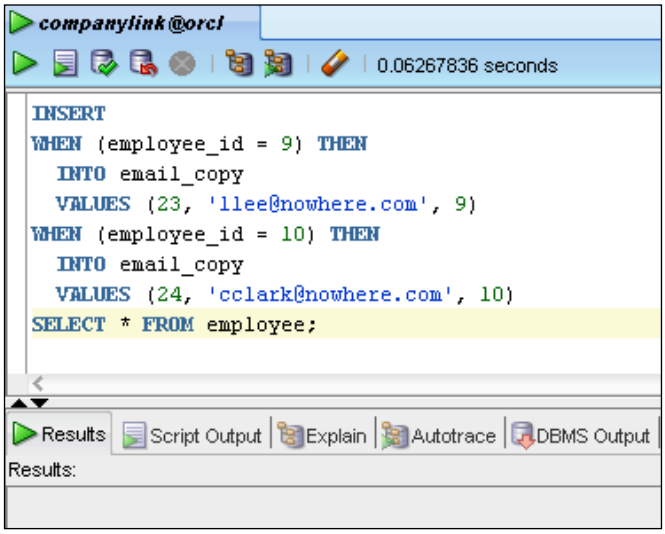
In this statement, we use a `SELECT` statement to copy the values from the `email` table, row by row, and insert them into the table called `email_copy`. In order for this statement to succeed, it is vital that the `email` and `email_copy` tables have the same order and number of columns and that their datatypes match. Failure to do so will result in either an error or data insertion into the wrong columns. This type of multi-row insert is also referred to as an `INSERT..SELECT` statement, because of its use of the `SELECT` statement to retrieve the rows to be inserted. If we had selected all the rows from the `email_copy` table before the previous `INSERT..SELECT` statement, we would have retrieved zero rows. The `email_copy` table (by design) was empty. Now if we select the rows, we see rows that are identical to those in the `email` table, as shown in the following screenshot. This is essentially a method to copy rows from one table to another.



Conditional Inserts—INSERT...WHEN

Broadly speaking, we can say that single-row inserts use the simple `INSERT INTO` clause. As we saw in the previous section, multi-row inserts use the `INSERT... SELECT` format. We conclude our examination of `INSERT` statements with a look at a final type that uses a new clause—the `INSERT WHEN`. An `INSERT WHEN` clause invokes a **conditional insert** statement. It inserts certain rows based on a given condition.

We can think of a conditional insert as an `INSERT` statement paired with a conditional `IF...THEN` statement. A conditional `INSERT` statement is shown in the following example:



To understand this statement, we look first at the last line—the `SELECT` statement. This portion of the statement drives the conditions that are used. The statement will first select all rows from the **employee** table. Whenever it encounters a row where the **employee_id** is **9**, it will insert a row into the **email_copy** table with the values specified in the first condition; namely, **23**, **'llee@nowhere.com'**, and **9**. Likewise, when it encounters a row where **employee_id = 10**, it will insert a row with the values specified in the second condition. Since there is one occurrence of the value `9` in the `employee` table and one occurrence of the value `10`, this statement will insert two values into the `email_copy` table. The results are shown in the following screenshot. Compare this to the rows in the `email` table to see the difference.

companylink@orcl

0.04119937 seconds

```
select * from email_copy;
```

Results | Script Output | Explain | Autotrace | DBMS Output

Results:

	EMAIL_ID	EMAIL_ADDRESS	EMPLOYEE_ID
1	1	jjohnson@companylink.com	1
2	2	jjohnson@gmail.com	1
3	3	mwilliams@companylink.com	2
4	4	landerson@companylink.com	3
5	5	drobinson@companylink.com	4
6	6	mgarcia@companylink.com	5
7	7	mgarcia@hotmail.com	5
8	8	hharris@companylink.com	6
9	9	kwhite@companylink.com	7
10	10	kwhite@yahoo.com	7
11	11	dperez@companylink.com	8
12	12	llee@companylink.com	9
13	13	llee@gmail.com	9
14	14	llee@yahoo.com	9
15	15	cclark@companylink.com	10
16	16	gmoore@companylink.com	11
17	17	chall@companylink.com	12
18	18	srodriguez@companylink.com	13
19	19	klewis@companylink.com	14
20	20	klewis@gmail.com	14
21	21	gtaylor@companylink.com	15
22	22	lthomas@companylink.com	16
23	23	llee@nowhere.com	9
24	24	cclark@nowhere.com	10

The `INSERT WHEN` is a less common, more advanced `INSERT` statement. It is not covered on the certification exam, but is included here for completeness as an example of a conditional insert.

Modifying data with UPDATE

Next in our examination of the subject of data manipulation is the ability to change existing database data. This section looks at a number of the common methods to update data within our database.

Understanding the purpose and syntax of the UPDATE statement

The first of our DML statements allowed us to insert new data into a table. However, often the data we're interested in already exists in the table and simply needs to be altered in some way. In these circumstances, we need to modify the data, not insert it. In such situations, we use an `UPDATE` statement. Its syntax tree is shown as follows:

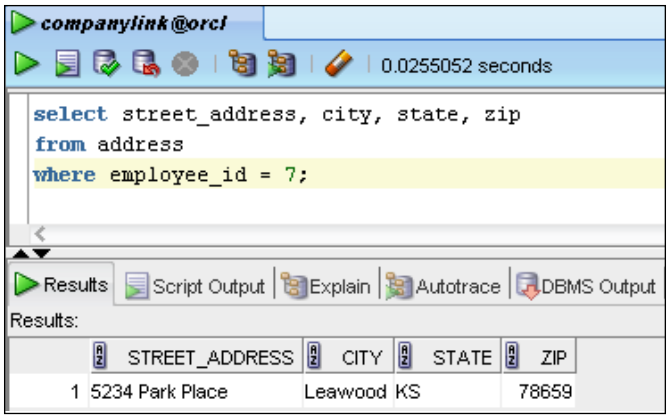
```
UPDATE {table_name}
SET {column_name} = {value2}
[WHERE {column_name} = {value1}];
```

Using the `UPDATE` statement, we modify the data in a particular column based on a condition—where a column name equals some value that we provide.

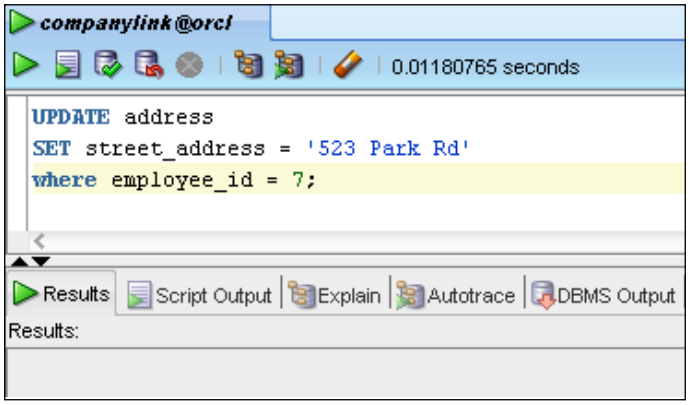
Writing single-column UPDATE statements

A situation has occurred at Companylink. A number of employees participating in the Companylink social networking site have changed their addresses, but these changes were never recorded in our database. Additionally, some of the addresses were recorded incorrectly when they joined Companylink. Because of this, a number of employees on the mailing list are not getting their written privacy notifications. It's up to us as the SQL programmers to fix this situation. To do it in the most efficient manner possible, we'll use the `UPDATE` statement.

The first of our corrections will be to fix the incorrectly recorded address for Ken White, whose employee ID number is 7. The address shown for Ken is 5234 Park Place, Leawood KS, 78659. His correct address is 523 Park Rd, Leawood KS, 78659. Our next example shows us Ken's incorrect address, for reference. The following screenshots demonstrate the way to use an `UPDATE` statement to correct the problem:



Next, we issue an `UPDATE` statement to change Ken's address to the correct value.

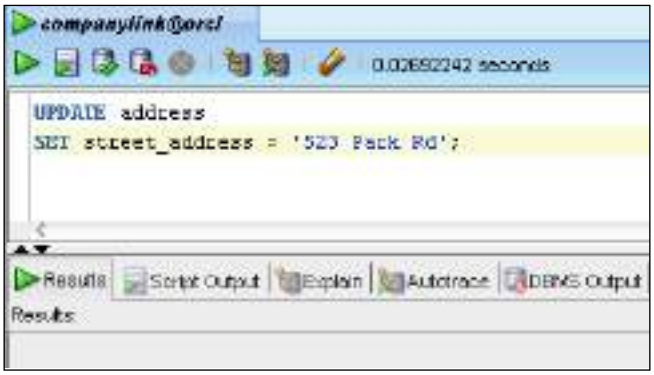


To write this statement, we first identify the condition we will use to determine which rows are updated. In our example, we're targeting the row or rows where **employee_id** equals **7**. In our case, this is only one row. Once that row has been identified, we construct our statement to alter the value for **street_address**, which is specified by our **SET** clause, and change it to the string value '**523 Park Rd**'. Once the statement is executed, SQL Developer reports back to us that one row was updated, in the lower-left corner of the screen.

We can see the change in Ken's address using the following query:

It is important to notice several facts about our `UPDATE` statement. First, the `WHERE` clause specifies a condition—any condition we choose. In our case, we defined this condition as **where employee_id = 7**. We could, however, choose to assign a condition based on another column. One very straightforward condition we could have used would be the clause `where street_address = '5234 Park Place'`. This would locate the row where this address condition was true and update the data accordingly. However, we would have to be certain that no other employee had a street address of '5234 Park Place', perhaps in another city. If there were two occurrences of that street address, both would be changed. Oracle updates the data based on a strict interpretation of the condition, so you must be careful when assigning it.

The second, and probably most important, fact to remember when using `UPDATE` statements is that *it is crucial to specify a condition*. Assume that we re-wrote our `UPDATE` statement incorrectly, as shown in the next screenshot. Do not run this statement—it will modify all of the column data.



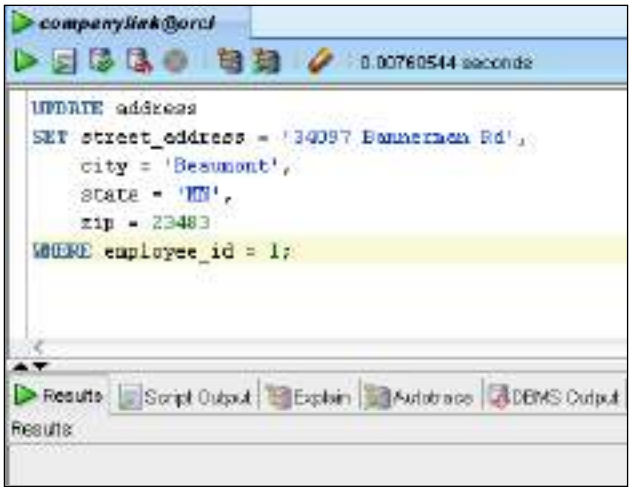
In this statement, we've instructed Oracle to update the address table as before, setting `street_address` to the indicated value. However, we have neglected to specify the `WHERE` clause that forms our condition. As a result, Oracle will update the `street_address` column in the table, *changing each street_address in the table to the specified value*. In short, if we run this statement without a limiting condition, every employee will have a street address of **'523 Park Rd'**. Unless this is what we want, we must include the condition that limits the rows to be updated.

SQL in the real world

Omitting the limiting condition when running DML statements is one of the most common mistakes that is made by new SQL coders. The results of such a mistake can be disastrous if done on a production table. However, in some circumstances, running an update on every row value for a column may be completely legitimate, such as the goal of updating a particular date column with the current date. Omitting the limiting condition isn't always wrong. Just remember to use the `WHERE` clause when it is needed.

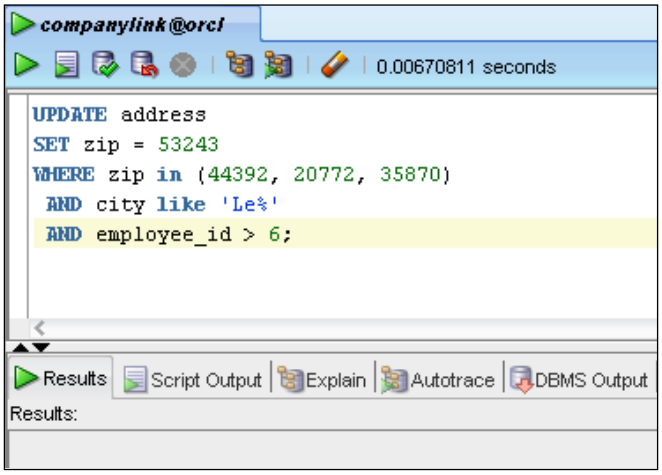
Multi-column UPDATE statements

For our second task in correcting our Companylink addresses, an employee, James Johnson, has moved to an entirely new address in a different city. However, his row values for `employee_id` and `address_id` do not need to change. This leaves us with four column values to update: `street_address`, `city`, `state`, and `zip`. We could accomplish this by running four separate `UPDATE` statements, each specifying a different `SET` clause. However, the best approach is to use our `UPDATE` statement to change multiple column values at once, as shown in the next screenshot:



In this example, we identify James Johnson as having an employee ID number of **1**. This forms our limiting condition. Next, we need to instruct Oracle to update four columns: `street_address`, `city`, `state`, and `zip`, and change them to the values we've provided. To do this, we specify each column and its new value, separated by commas. Our example places them on different lines for greater readability, although this isn't syntactically required. Using this multi-column `UPDATE`, we can update any column, provided that it exists in the table.

Although it is common to use conditions of equality when forming limiting conditions, it is not required. Any of the conditions we saw in *Chapter 3*, *Using Conditional Statements* can be used as a condition in an `UPDATE` statement, including conditions of non-equality, range, list, set conditions, and even Boolean conditions. The statement in the following screenshot combines a number of these types of conditions—list conditions, Boolean conditions, and pattern-matching.



Removing data with DELETE

Lastly in this section, we examine the use of DML statements to remove data from the database.

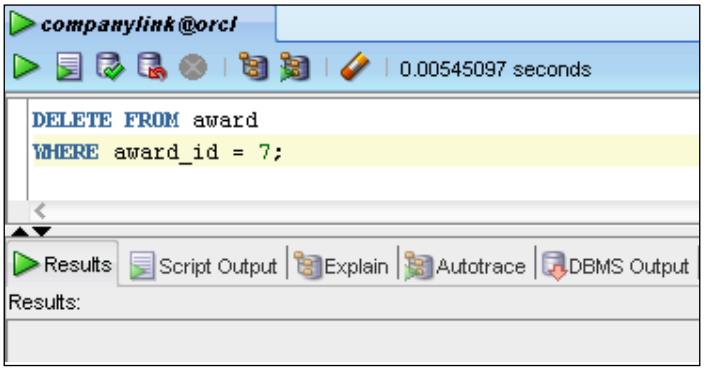
The purpose and syntax of the DELETE statement

As important as it is to be able to add and change data, sometimes data needs to be removed from a database. For example, say that a company keeps an *opt-in* mailing list. When a customer *opts-in* to receive mailings from the company, their information is placed into a table called `mailing_list`. If the customer decides that they no longer want to receive information from the company, they can *opt-out* of the mailings. In such a situation, the information should be removed from the `mailing_list` table. In SQL, we remove data from a table using the `DELETE` statement. Its syntax tree is shown as follows:

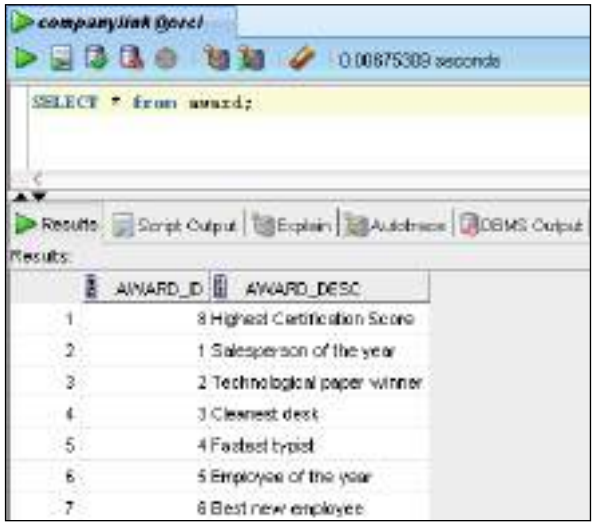
```
DELETE FROM {table_name}
[WHERE {column} = {value}];
```

Deleting rows by condition

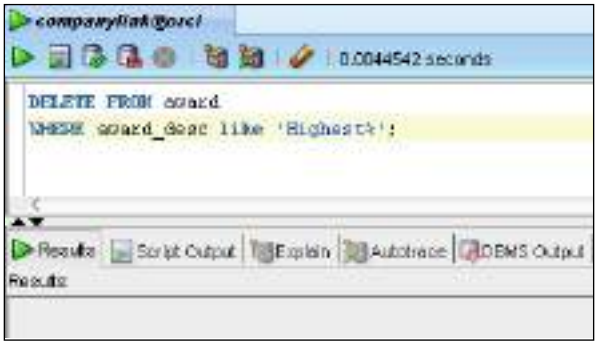
In SQL, the `DELETE` statement is the logical opposite of an `INSERT` statement. We use it to remove rows from a table. However, unlike the `INSERT` statement and similar to the `UPDATE`, we specify a condition when using a `DELETE`. This condition, denoted with a `WHERE` clause, will determine which rows are deleted from the table. Our first example of a `DELETE` statement is shown in the following screenshot, which is the logical opposite of the `INSERT` shown in the previous `INSERT` statement:



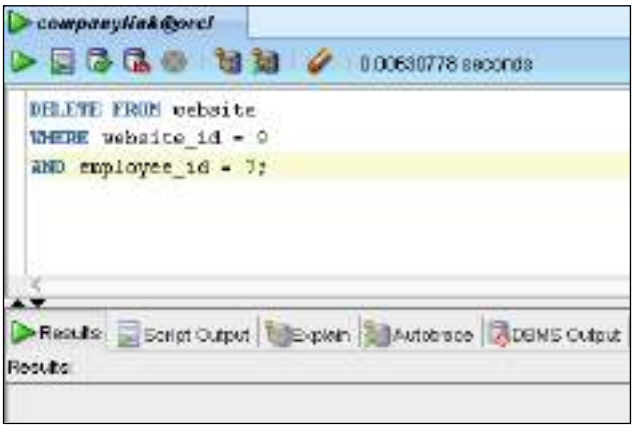
In this example, we designate the `award` table as the target for our deletion. We next specify a `WHERE` clause that indicates rows with an **award_id** value of **7** will be deleted. Since our Companylink `award` table has only one row with an `award_id` value of `7` (the one we inserted), only one row will be deleted. We see the result of the `DELETE` in the following query. For clarity, you can compare this to the previous query to see the table rows before the `DELETE` occurred.



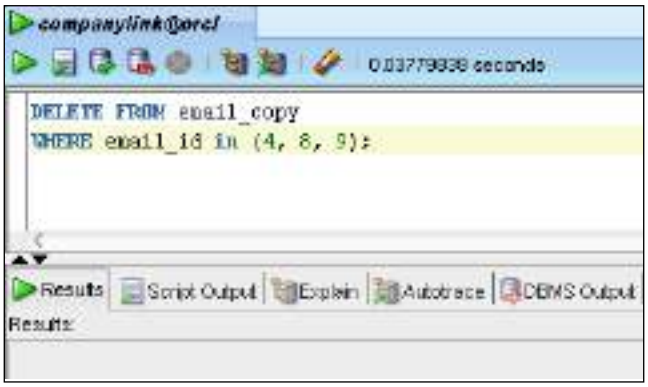
Because the `DELETE` statement removes an entire row, we do not make use of a named column list, as with the `INSERT` statement. We simply identify a value or values that indicate which rows should be removed. Like the `UPDATE` statement, however, we can make use of the numerous types of conditions found in `WHERE` clauses. The next several examples demonstrate some of the different types of conditions we can couple with our `DELETE` statements. First, we demonstrate `DELETE` operations using pattern matching with `LIKE`.



Next, we look at the use of Boolean conditions, specifically the AND.



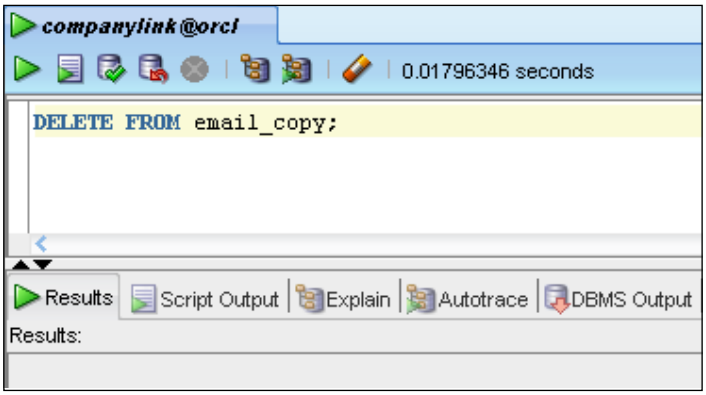
Lastly, we examine the use of set operations as conditions for a DELETE statement.



Deleting rows without a limiting condition

As with the UPDATE statement, it is crucial to remember that any DELETE statement executed without a WHERE clause *will delete all the rows in the table*. It is extremely important to be able to accurately identify the rows you wish to delete. Failing to do so can have unforeseen consequences. The following example demonstrates the results of issuing a DELETE without a limiting condition. If you select the rows from the email_copy table after running the DELETE statement below, you will see that they have all been removed.

Again, sometimes this is what you're trying to achieve. Just remember—use the `DELETE` statement with the utmost care.

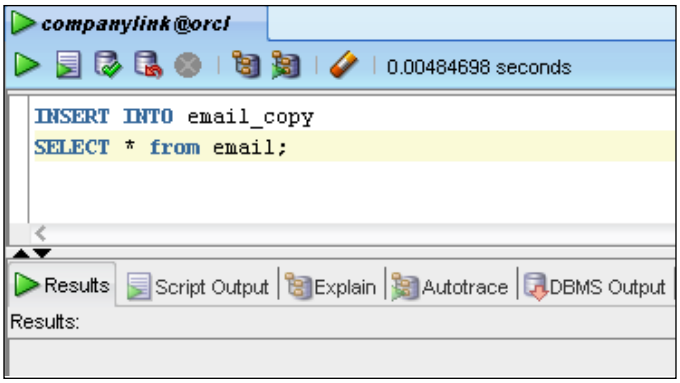


Removing data unconditionally with TRUNCATE

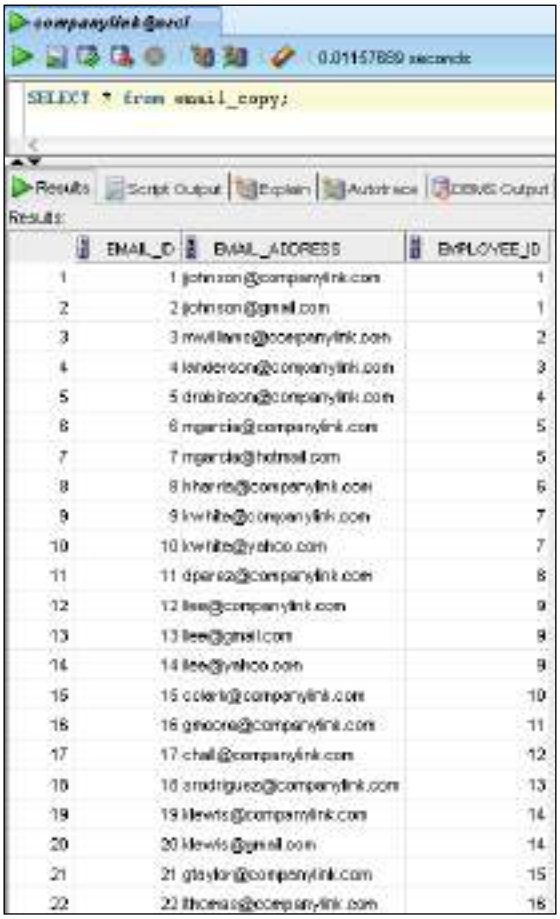
For our final look at data manipulation, we examine a statement that is slightly different than the three DML statements we have seen in this chapter. Although like the `DELETE` statement it removes row data, **TRUNCATE** is not a part of the DML family at all. It is, in fact, a member of the DDL sublanguage, which stands for **data definition language**. DDL is generally used to structurally alter a database object, such as a table, in a fundamental way. We will see many examples of DDL in later chapters, but for now, we look at the way `TRUNCATE` operates on data.

The `TRUNCATE` command is used to unconditionally remove all the rows in a table. Structurally, the `TRUNCATE` statement deletes data from a table in a way that is fundamentally different than the `DELETE`. For reasons we will revisit in the next section, a `TRUNCATE` command can remove all the rows in a table almost instantly, while the `DELETE` may take a significant amount of time, depending on the number of rows being deleted. Thus, when you need to unconditionally remove all the rows in a table, it is generally better to utilize a `TRUNCATE` statement.

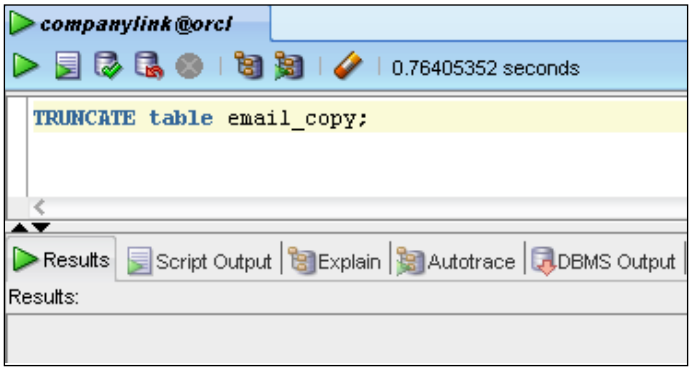
In order to demonstrate `TRUNCATE`, we will walk through a number of steps to repopulate our `email_copy` table, then remove the data using `TRUNCATE`. These steps are shown in the following four screenshots. The first step populates the `email_copy` table using data from the `email` table.



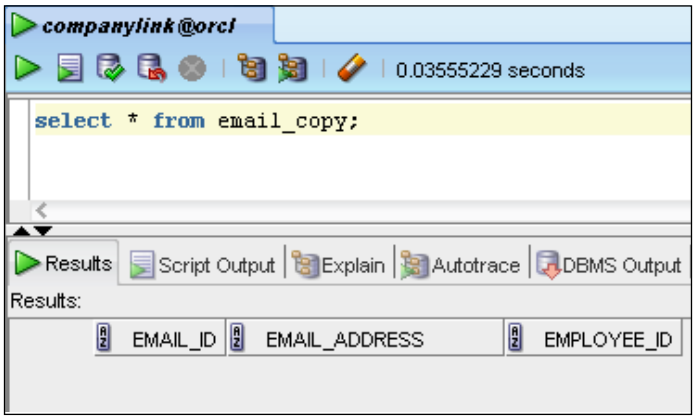
Next, we verify that the data is actually present using a `SELECT` from the `email_copy` table.



We then execute the `TRUNCATE` statement that unconditionally removes all data from the table. The `TRUNCATE` operation is nearly instantaneous, in contrast to `DELETE`s from large tables that can take considerable time.



Finally, we query the table again to see that all the data is, in fact, removed.



Transaction control

What would you say if I told you that none of the DML statements that you've executed in this chapter, `INSERT`s, `UPDATE`s, and `DELETE`s, actually did anything to change data in the database? As much of a surprise as it is, it is actually true. The use of DML statements comes with a caveat: they must be used in conjunction with transaction control statements.

Transactions and the ACID test

Transaction control is the act of manipulating the timing of events called transactions. In relational database theory, a **transaction** is a discrete unit of work within a database. Transactions allow groups of statements to be executed together, to allow for correct recovery in the event of failure. Transactions also represent data **concurrency**, a process by which multiple users can manipulate data without the fear their data will be modified with unintended consequences. In essence, concurrency plays the role of *traffic cop* for multiple users, allowing them to modify data according to a set of rules. These rules are represented by the acronym **ACID**: **A**tomicity, **C**onsistency, **I**solation, **D**urability.

1. Atomicity ensures the completeness of a transaction by enforcing the *all or nothing* rule. With an atomic transaction, all statements must either succeed or all must fail.
2. Consistency states that the data returned from a query will be consistent with the state of the data when the transaction began. If the data being selected is also being changed by other users, the data results will appear as they were when the transaction was executed.
3. Isolation enforces the rule that the results of any query against data in the process of being changed must display the unchanged data until a transaction completes. In short, data changes must be hidden until a transaction finishes.
4. Durability refers to the guarantee that, once committed, transactions cannot be *lost*. Once a durable transaction is committed, its results are seen as *real* and cannot be reversed.

The ACID test deals with the ability of an RDBMS to handle data in a state of flux, or change, in a reliable way. It prevents hazards such as partially changed data, false results, and data being changed in an unintended way. The ACID test is crucial to any RDBMS that claims to have the ability to handle large amounts of data and large numbers of concurrent users. Like most of the database systems available today, Oracle's rules of transaction control pass the ACID test.

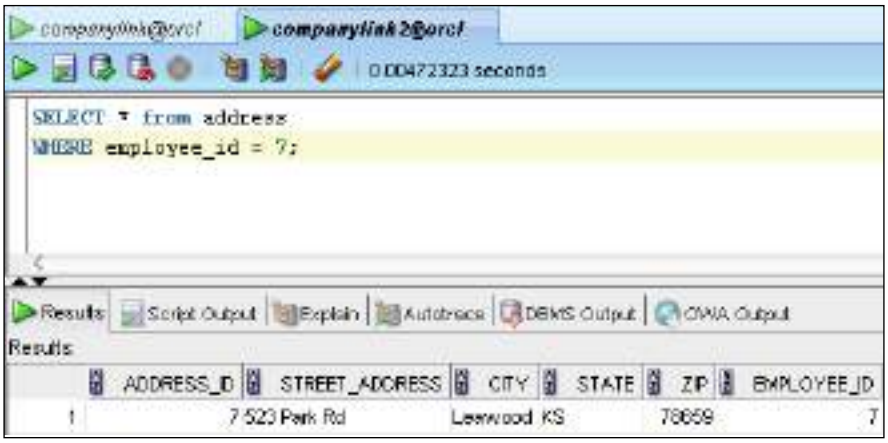
In an Oracle database, transaction control is achieved using statements that belong to the sublanguage called TCL, or Transaction Control Language. TCL statements allow users to begin a transaction and either end it successfully or revert back to the state of the data prior to the time the transaction began. There are three primary commands in the TCL sublanguage—COMMIT, ROLLBACK, and SAVEPOINT.

Completing transactions with COMMIT

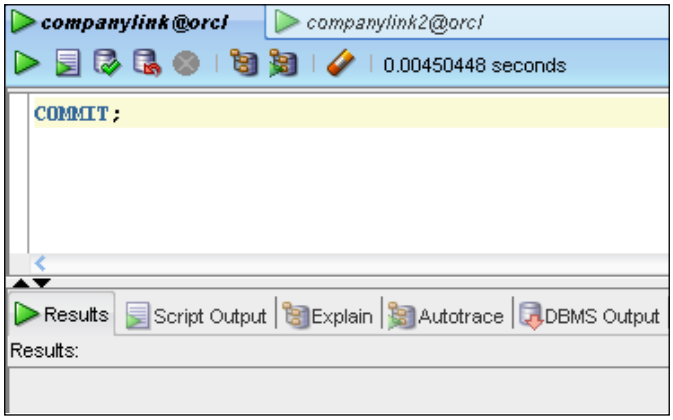
In Oracle SQL, the `COMMIT` statement is used to signify the end of a transaction. The transaction begins with the first DML statement and does not end until a `COMMIT` is executed. In the time between the beginning of the transaction and the `COMMIT`, the data being changed is not visible to any other user sessions, in accordance with the ACID rule of Isolation. A transaction can consist of one or many `INSERT`, `UPDATE`, and `DELETE` statements. We will demonstrate the isolating nature of transaction control in the next several screenshots. You can type in these examples from your `Companylink` database, but to actually see the effect at work, you will need to have two separate connections into your database. The first step is to begin a transaction, as shown with the `UPDATE` statement in the following example:

Here, the transaction has begun, but no `COMMIT` has been executed, indicating that the transaction is not yet complete. Because of this, the session that changed the data can view the changes, as shown in the next screenshot:

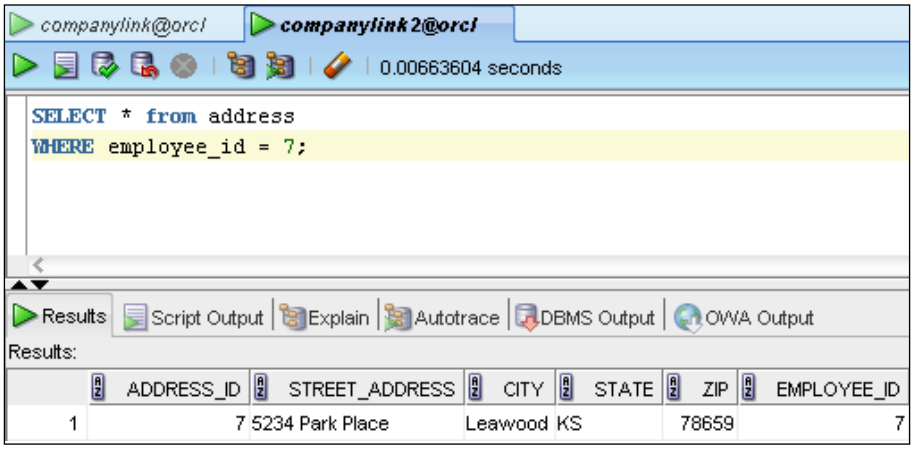
However, if we open a second connection in SQL Developer, our next example (notice the two tabs at the top) shows us that the new session cannot see the data changes made by session #1, due to the isolating nature of transaction control. Notice how the `street_address` column in the next example differs from the previous one.



As you can see, the `street_address` shown in the second session is `'523 Park Rd'`, the "before image" of the data prior to a commit. Next, we execute a `COMMIT` statement in the original session, the one that actually executed the `UPDATE` statement, as shown in the following example. This indicates that the transaction is complete.



Finally, we run the query from the second session again, as shown in the next screenshot. It shows the state of the data from the `address` table from the perspective of the second session following a `COMMIT` in the first session.



Now we see that since the transaction is complete, both sessions now see the same data.

SQL in the real world

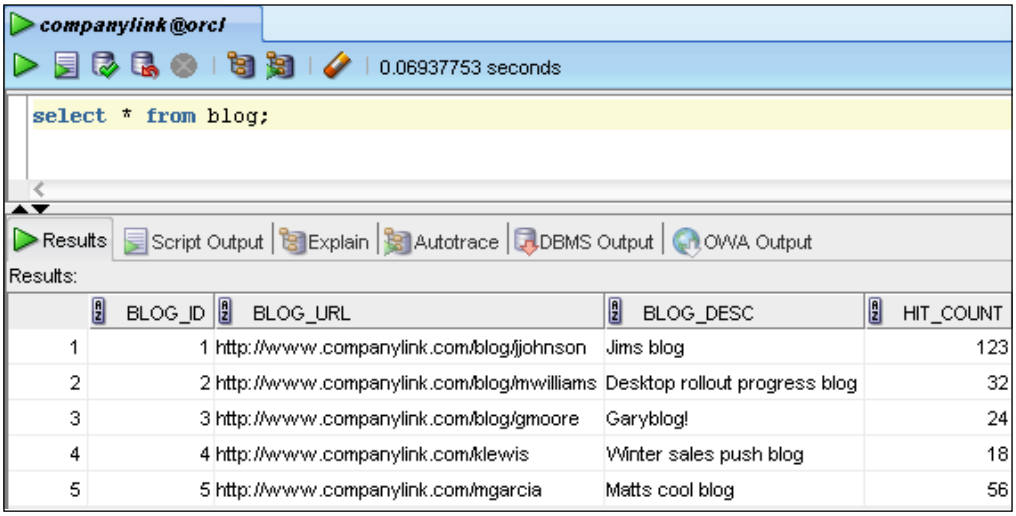
In real-world coding situations, special care should be taken when deciding how often to issue the `COMMIT` statement. If you commit too often, your performance will suffer. If you execute millions of statements without a commit, you can cause locking problems that affect the performance of other users. A good SQL programmer needs to strike a balance between the two.

Undoing transactions with ROLLBACK

We've stated thus far that transactions begin with a DML statement and end with a `COMMIT`, which completes the transaction successfully. You may ask why that should even be necessary. Why shouldn't a transaction be considered complete whenever the DML statement is finished? There are several reasons for this level of control, but one of the most beneficial ones is the ability to *undo* a transaction. If you make a mistake, such as running a `DELETE` statement without a limiting condition, you can undo it with another TCL statement—`ROLLBACK`.

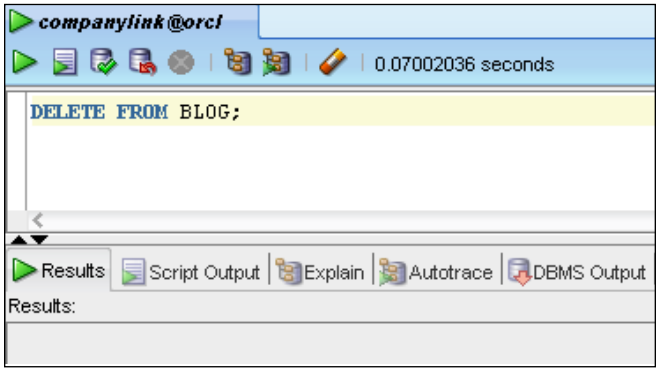
The `ROLLBACK` statement does exactly what it implies—it rolls back a transaction to its original state—provided that you have not already issued a `COMMIT`. In Oracle, the data involved in a DML statement, such as the values in an `INSERT`, are not

directly applied to the table until the transaction is complete. This is the reason that, while you can see your DML changes, other sessions cannot. Prior to the `COMMIT`, the values are placed in a kind of *holding area* known as the **Undo tablespace**. This area holds the pre-committed version of the data until the transaction is complete. Since the undo space has our data in its original state, it can be moved back and re-applied to the table if necessary using a `ROLLBACK` statement. The next several screenshots demonstrate the use of the `ROLLBACK` command. First, we verify that the data from the blog table is present.



	BLOG_ID	BLOG_URL	BLOG_DESC	HIT_COUNT
1	1	http://www.companylink.com/blog/jjohnson	Jims blog	123
2	2	http://www.companylink.com/blog/mwilliams	Desktop rollout progress blog	32
3	3	http://www.companylink.com/blog/gmoore	Garyblog!	24
4	4	http://www.companylink.com/klewis	Winter sales push blog	18
5	5	http://www.companylink.com/mgarcia	Matts cool blog	56

Next, we execute a `DELETE` without any limiting condition.

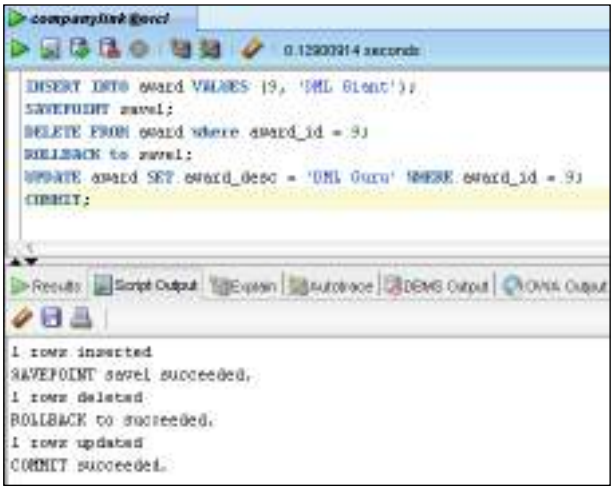


As a result of the unrestricted `DELETE`, all data has been removed from the blog table. However, take note that no `COMMIT` statement was issued.

To correct our mistake, we issue the `ROLLBACK` command.

We then see that the data is restored.

In Oracle, we can even partially rollback a statement using the `SAVEPOINT` command. A `SAVEPOINT` is a named breakpoint or marker that indicates a place to which a `ROLLBACK` can occur. While a `SAVEPOINT` is used in transaction control, the statement itself does not end the transaction in the way that `COMMIT` and `ROLLBACK` do. Consider the set of statements in the following screenshot. Because this example uses multiple statements, you must execute it by clicking the **Run Script** button just to the right of the green arrow **Execute Statement** button in SQL Developer. You can also invoke Run Script using the *F5* key.



Let's step through this statement one line at a time. Our first statement inserts values into the `award` table. Next, we place a `SAVEPOINT` called `save1` into our transaction. We then execute a `DELETE` statement that removes the row we inserted. However, our next statement, `ROLLBACK` to `save1`, will rollback the `DELETE` so that it never occurred. Finally, we update the `award` table, changing the value for `award_desc`, and then issue a `COMMIT`. We see the final state of the data in the following query:

SQL in the real world

Be aware that a number of SQL tools available today use an **AUTOCOMMIT** feature. When AUTOCOMMIT is turned on, every DML statement will commit automatically. With some tools, such as SQL Developer, you must enable the feature. Others utilize it by default. Be mindful of this fact when issuing DML statements. If you want to stick to the SQL standard, leave AUTOCOMMIT off.

DELETE and TRUNCATE revisited

As a final word on transaction control, we want to quickly revisit the difference between the `DELETE` and `TRUNCATE` statements. We mentioned earlier in the chapter that a `TRUNCATE` command will remove all the rows in a table and can do it much faster than a `DELETE`. Now that we've seen how transaction control works, we can see why. Since `DELETE` is a DML statement, we know that any deleted rows are first put into the undo space in case of a `ROLLBACK`. If we are deleting a large number of rows, it can take time to do this operation. The `TRUNCATE`, on the other hand, is a DDL statement—it requires no `COMMIT` in order to remove the rows, and thus puts no data into undo space. This leads us to two conclusions. First, the `TRUNCATE` can remove data quickly since it writes no undo data. Second, since the `TRUNCATE` is a DDL statement, we cannot issue a `ROLLBACK` after a `TRUNCATE` to retrieve the original data. The `TRUNCATE` command is fast, but it is also irreversible.

Recognizing errors

To wrap up this chapter, it's time to address what happens when you receive an error in SQL Developer. By this time, it's likely that you may have made a mistake in typing a statement and received an error. Errors can be frustrating, but debugging your code is an essential part of learning to write SQL. To generate an example error, we incorrectly write the following SELECT statement:

```
SELCT * FROM award;
```

We receive this error in SQL Developer.

Examine the error generated previously. The window that pops up indicates an error was encountered. It lists an error number, **ORA-00900**, and an error message, **"invalid SQL statement"**. The rest of the information is not relevant to this discussion. Every SQL error in Oracle has a designated error number and error message. Many different kinds of statements generate different errors. The following screenshot displays the type of error message received when you attempt to change a numeric value into a character one with the following statement. The `award_id` is a numeric column, so attempting to set it to the string value, `'Hello'`, results in an error.

```
UPDATE award SET award_id = 'Hello' WHERE award_id > 3;
```

The previous screenshot shows an example of the error returned from referencing an incorrect column name. The actual column name is `award_id`, but our `WHERE` statement refers to it as `award_ident`.

```
UPDATE award SET award_id = 3 WHERE award_ident = 1;
```

Like most implementations of the language, Oracle's SQL is very strict. Any deviation from correct syntax and semantics will produce an error. Some common conditions that generate errors include:

- Misspellings of SQL clauses, such as `SELECT` or `INSERT`
- Improperly constructed SQL statements
- Mismatched data types
- Violation of primary or foreign keys
- Insufficient object privileges
- Violation of NOT NULL constraints
- Improperly referenced table or column names

When you encounter an error, simply step through your statement slowly, checking for any misspellings, missed punctuation such as commas, or incorrect table names. Be patient and refer to documentation when necessary. Debugging is a skill that is acquired over time.

Summary

In this chapter, we've added the power of DML to our SQL abilities. We learned to add, modify, and remove data from relational tables using `INSERT`, `UPDATE`, and `DELETE`. We've learned the importance of adding limiting conditions to our `UPDATE` and `DELETE` statements. We've learned about the concept of transactions and used `COMMIT`, `ROLLBACK`, and `SAVEPOINT` to achieve transaction control. Lastly, we've looked at the types of error messages generated by Oracle.

Certification objectives covered

- Describe each data manipulation language (DML) statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Control transactions

At this point in the book, we've learned the basics of retrieving and manipulating data with SQL statements. However, up to now, everything we've seen has focused on statements that work with a single table. In examples where we used `SELECT`, we did so only with a single table. In the next chapter, we begin to broaden our abilities. We'll learn how to take data from multiple tables and combine it in useful ways by joining tables.

Test your knowledge

1. Which of the following terms does not apply to the CRUD model of persistent storage?
    a. `Create`
    b. `Undo`
    c. `Destroy`
    d. `Read`
2. Which of the following is not a DML statement?
    a. `INSERT`
    b. `UPDATE`
    c. `COMMIT`
    d. `DELETE`

3. Which of these terms is a DML statement that allows you to add rows to a table?

   a. `CREATE`

   b. `INVOKE`

   c. `DELETE`

   d. `INSERT`

4. Given the structure of the branch table shown below, which of these `INSERT` statements uses correct positional notation?

```
BRANCH_ID NUMBER(10)
BRANCH_NAME VARCHAR2
DIVISION_ID NUMBER(10)
```

   a.
```
INSERT INTO branch (branch_name, division_id, branch_id)
 VALUES ('Executive', 7, 14);
```

   b.
```
INSERT INTO branch
VALUES ('Executive', 7, 14);
```

   c.
```
INSERT INTO branch (branch_name, division_id, branch_id)
VALUES (14, 'Executive', 7);
```

   d.
```
INSERT INTO branch
VALUES (14, 'Executive', 7);
```

5. Which of these INSERT statements uses correct named column notation?

   a.
```
INSERT INTO branch (branch_name, division_id, branch_id)
VALUES ('Executive', 7, 14);
```

   b.
```
INSERT INTO branch
VALUES ('Executive', 7, 14);
```

   c.
```
INSERT INTO branch (branch_name, division_id, branch_id)
VALUES (14, 'Executive', 7);
```

   d.
```
INSERT INTO branch
VALUES (14, 'Executive', 7);
```

6. If the following statement was executed against the branch table in the `Companylink` database, what value would be inserted into the `division_id` column?

```
INSERT INTO branch (branch_name, branch_id)
VALUES ('Supervisory', 10);
```

   a. Supervisory
   b. 10
   c. Null
   d. 0

7. If the following statement was executed against the `Companylink` database, which of the following columns would not be present in the `address_copy` table?

```
INSERT into address_copy
SELECT * FROM address;
```

   a. `city`
   b. `street_address`
   c. `zip`
   d. `division_id`

8. Which of the following UPDATE statements is syntactically correct?

   a.
```
UPDATE email SET email_address = 'donperez@companylink.com'
WHERE email_id = 11;
```

   b.
```
UPDATE email WHERE email_id = 11
SET email_address = 'donperez@companylink.com';
```

   c.
```
UPDATE email WHERE email_id = 11;
```

   d.
```
UPDATE email
SET email_address TO 'donperez@companylink.com';
```

9. Consider the following row in the blog table containing the values in the column order of the table.

```
6, 'http://www.companylink.com/testpage', 'Test Description',
20
```

Which of these columns is unchanged if the following UPDATE statement is executed?

```
UPDATE blog
SET blog_id = 7,
  blog_desc = 'Test Description2',
  hit_count = 30
WHERE blog_id = 6;
```

a. `blog_id`

b. `blog_url`

c. `blog_desc`

d. `hit_count`

10. Refer to the branch table in your `Companylink` database. If the following statement was executed, how many rows would be deleted?

```
DELETE FROM branch
WHERE division_id = 3;
```

a. 1

b. 2

c. 3

d. 12

11. Refer to the branch table in your `Companylink` database. If the following statement was executed, how many rows would be deleted?

```
DELETE FROM division;
```

a. 1

b. 4

c. 6

d. 0

12. Which of the following `DELETE` statements is syntactically correct?

a. `DELETE * FROM branch WHERE branch_id = 5;`

b. `DELETE FROM branch WHERE branch_id = 5;`

c. `DELETE WHERE branch_id = 5 FROM branch;`

d. `DELETE * WHERE branch_id = 5 FROM branch;`

13. Which of the following represents the proper syntax for a TRUNCATE statement?
    a. `TRUNCATE branch;`
    b. `TRUNCATE table branch;`
    c. `TRUNCATE table branch where branch_id is null;`
    d. `TRUNCATE FROM table branch;`

14. Which of these terms is not a part of the transaction control acronym ACID?
    a. Atomicity
    b. Isolation
    c. Commit
    d. Durability

15. Consider the following set of statements. What are the values for the row having `branch_id` equal to `14` for `branch_id`, `branch_name`, and `division_id` respectively, at the end of the statement?

```
INSERT INTO branch VALUES (14, 'Research', 7);
COMMIT;
UPDATE branch SET division_id = 8 WHERE branch_id = 14;
SAVEPOINT saveit;
DELETE FROM branch WHERE branch_id = 14;
ROLLBACK to saveit;
UPDATE branch SET branch_name = 'R and D' WHERE branch_id = 14;
COMMIT;
```

    a. 14, 'Research', 7
    b. 14, 'R and D', 8
    c. null, null, null
    d. An error is returned

5
Combining Data from Multiple Tables

The ability to retrieve data from a table is an absolutely essential element of learning SQL. However, real-world requirements often demand the ability to select data from multiple tables concurrently and present it in a meaningful way. Herein lies the ability to join tables. This chapter will cover two distinct syntaxes for joins as well as multiple techniques for combining data from multiple tables.

In this chapter, we shall:

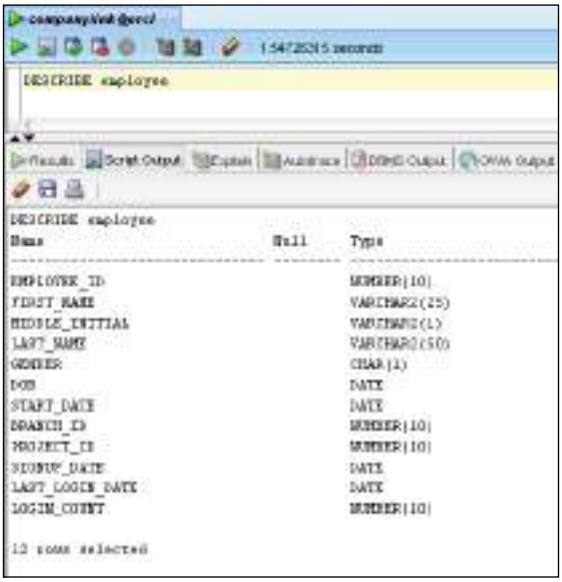
- Examine the concept of joining multiple tables together
- Join tables using ANSI-compliant join syntax
- Look at **n-1** join conditions
- Join tables using the new Oracle `join` syntax

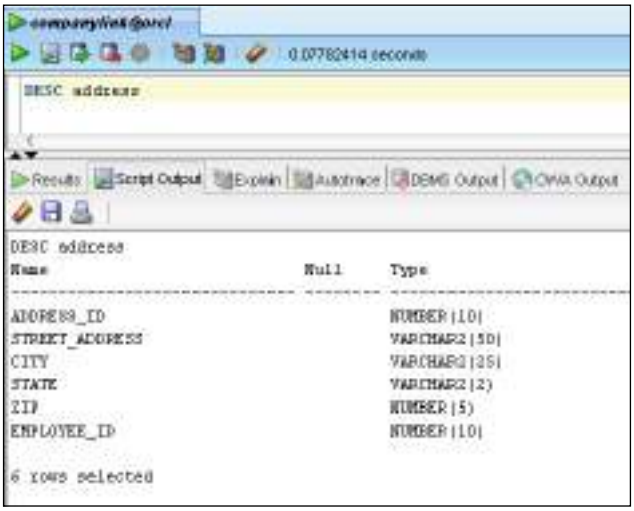
Understanding the principles of joining tables

So far, we have seen several ways to select data from tables. In *Chapter 2*, *SQL SELECT Statements*, and *Chapter 3*, *Using Conditional Statements*, we covered numerous ways to retrieve data and then limit its retrieval by condition. In *Chapter 1*, *SQL and Relational Databases*, we extensively discussed the concept of relational databases. In that chapter, we stated that what makes a relational database different from a flat file database are the relationships between entities, or in our case, tables. Up to now, we haven't seen the effect of these inter-table relationships. Each of our queries is applied only to one table. So, while each of these queries have been effective for our needs, those needs have never gone beyond the search for data from a single table. In short, if called on to do so, how would we pull data from multiple tables with a single query?

Accessing data from multiple tables

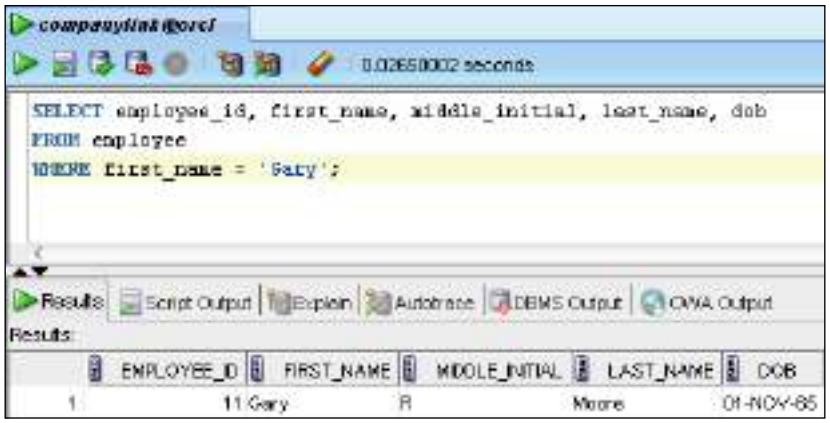
Consider the columns from two tables, shown in the following two screenshots. Here, we revisit the `DESCRIBE` command from *Chapter 2*, *SQL SELECT Statements*, which lists the columns that make up the `employee` table. In the second screenshot, we used the shortened version of the command, `DESC`, to do the same for the address table.



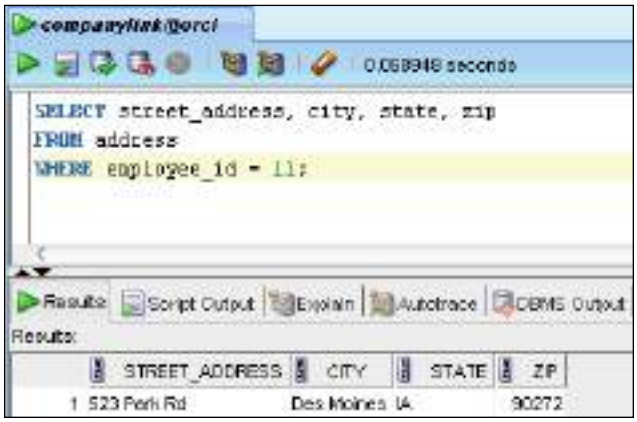


As we've said before, the columns in each table characterize different aspects of that table. The `employee` table contains information relevant to defining employees and the `address` table does the same for addresses. But, if that is the case, why do the `employee` and `address` tables both contain a column called `employee_id`? Obviously, the `employee_id` column is relevant to employee information, but why is it applicable to the `address` table? The sharing of columns such as `employee_id` between the `employee` and `address` tables is the key to understanding why an RDBMS is "relational". The relationships shared between tables are defined by the columns that they have in common.

To see this in action, consider this requirement for our `Companylink` database: *show me the full name, date of birth, and address for all employees named Gary*. One possible way to do this is with two queries. First, as shown in the following example, we use a query that selects name information from the `employee` table where the employee's first name is Gary.



From the results of this query, we see that the only employee named *Gary* is Gary R. Moore, who has an `employee_id` of `11`. This provides us with the name and date of birth, but not the address information. In fact, we cannot query the `address` table using the name **Gary**, since `first_name` is not a column in the `address` table. The only way for us to retrieve the correct address is to identify that Gary Moore has an `employee_id` of `11` and use that in a separate query of the `address` table, as shown in the following example:



The results show the address information for `employee_id` with value `11`, Gary Moore. We've found the information that was requested, but it has taken a long road to get there. What if the requirement was to display this information for *all* of the *Companylink* employees? Acquiring the information is possible, but doing so would require many queries. Fortunately for us, our `Companylink` database is a relational database. The `employee` and `address` tables have a relationship between each other. That relationship is formed by a common column—the `employee_id` column. Using that relationship, we can join the two tables.

The ANSI standard versus Oracle proprietary syntax

In Oracle, we actually have a choice between two distinct join syntaxes; the ANSI-compliant syntax and the newer Oracle proprietary syntax. The ANSI syntax is the standard supported by most relational database management systems, including Oracle. As we mentioned in *Chapter 1*, *SQL and Relational Databases*, the standards used in the SQL language are governed by the **American National Standards Institute** (**ANSI**) and the **International Organization for Standardization** (**ISO**). The ANSI syntax is the standard approved by ANSI and ISO. Its use is widespread. It is far more common to find ANSI joins in today's SQL than Oracle proprietary joins,

even in systems that use Oracle. Nevertheless, it is argued by some that Oracle's proprietary syntax is more intuitive and easier to understand. Both will be covered in this chapter. We'll revisit the Oracle standard later in the chapter.

SQL in the real world

Through the course of this chapter, you may find that you *like* one standard more than another. Just remember that in the real world, your choice of syntax may be limited by the coding standards of your organization. The Oracle syntax has yet to reach widespread acceptance. However, for the sake of the certification exam, it is crucial that you understand both. The Oracle `join` syntax is fairly heavily tested on the exam.

Using ANSI standard joins

In order to join two tables, we will utilize the basic structure of a `SELECT` statement; but, we must add a few qualifiers.

Understanding the structure and syntax of ANSI join statements

When we join two tables, we add a `WHERE` clause that qualifies that the common columns between the tables are equivalent. The following is the syntax tree for an ANSI-compliant join:

```
SELECT column1, column2, ...
FROM table1, table2
WHERE table1.common_column = table2.common_column;
```

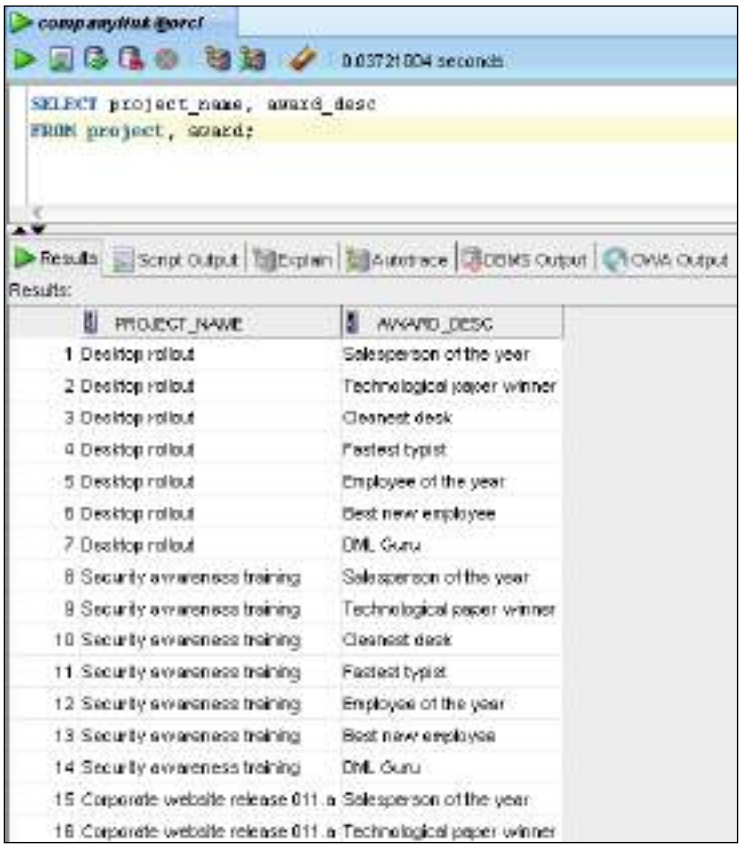
While the join syntax is similar in many ways to a typical `select` statement, we notice some differences.

- As you might expect, since we are selecting data from two tables, both table names are specified in the `FROM` clause, separated by commas
- The syntax of the `WHERE` clause is different from what we've previously seen
- In our `WHERE` clause, we specify the condition that the common column from the first table must be equal to the common column in the second

In constructing this part of the statement, it is important to have first identified the common column between the tables that forms the inter-table relationship. This equivalence forms the *bond* between the two tables.

Examining ambiguous Cartesian joins

Before we look at some examples of typical join statements, it is important to discuss one type of join that is considered undesirable in most circumstances. A **Cartesian join** is a join between two tables that omits the `WHERE` clause. The result is known as a **Cartesian product**. A Cartesian product is formed when every row of one table is joined to every row of another table. An example of this is shown as follows:



The resulting number of rows shown has been truncated for the sake of brevity. It is fairly easy to see what is happening without displaying all 40 rows that are returned. In a Cartesian join, the query returns the first row selected from the `project` table, in this case **Desktop Rollout**, and joins it to every row selected from the `award` table; first, **Salesperson of the year**, then **Technological paper winner**, and so on. It then moves to the second row returned from the `project` table, **Security awareness training**, and again joins each row from the `award` table. It continues in this manner until every row in the `project` table has been joined to every row in the `award` table.

The Cartesian product returned from a Cartesian join is generally considered undesirable because it has little meaning. Since no relationship has been established between the two tables based on a common column, the data from the `project` table does not relate in any logical way to the `award` table. Such joins are said to be ambiguous, since no row has any particular relationship to any other row.

It is important to remember that even if the two tables share a common column, that relationship must be specified in the `WHERE` clause. Failure to do so will result in a Cartesian product.

A Cartesian join is generally said to produce an *a times b* product, where *a* and *b* are the number of rows in the two tables. In the next screenshot, the `project` table has five rows, while the `award` table has eight rows. We can therefore say that the number of rows returned by the Cartesian product of the two tables will be 5 x 8, or 40, rows. In this way, we can predict the number of rows returned by any Cartesian join as being the number of rows in the first table times the number of rows in the second.

The eighth row in the `award` table was inserted in *Chapter 4*, *Data Manipulation with DML*. If you did not run the examples in that chapter, or if you have rebuilt your `Companylink` database since *Chapter 4*, *Data Manipulation with DML*, you will see seven rows instead. The resulting Cartesian product will thus be 5 x 7, or 35, rows.

SQL in the real world

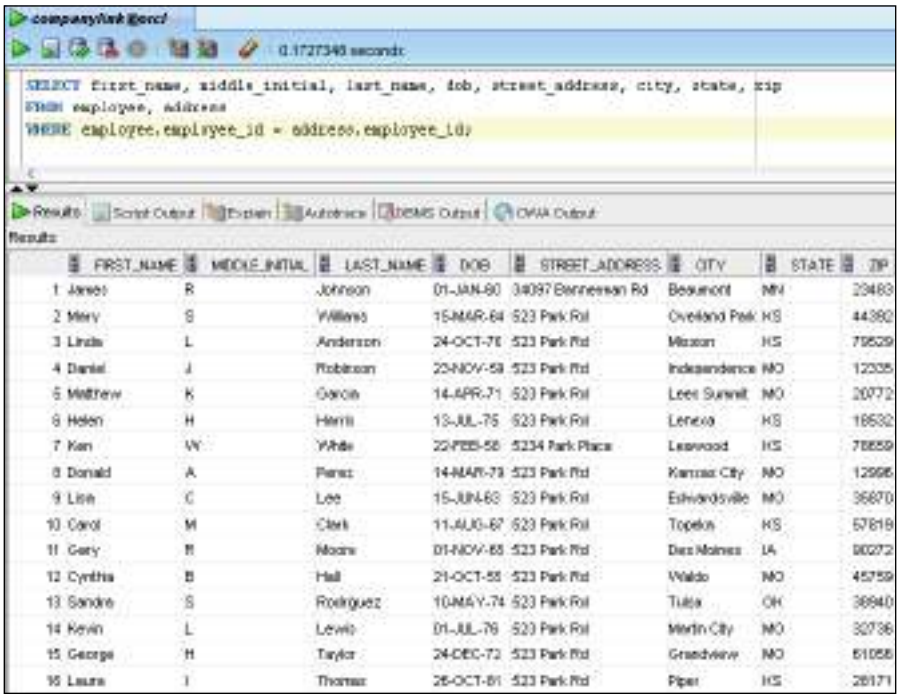
Another practical reason that Cartesian joins are considered undesirable is the immense strain they can put on a system from a performance perspective. While the 40-row Cartesian product from our example may not seem significant, consider two tables with one million rows each. The resulting number of rows from such a Cartesian product would be 1,000,000 x 1,000,000, or $1x10^{12}$, rows—one trillion rows. Such a mistake can cause excessive resource usage on your database system to the point of affecting other users. In fact, when tuning SQL statements, one of the most common examples of improper code to watch out for is Cartesian joins.

Using equi joins—joins based on equivalence

The core of the RDBMS is the relationships that are formed between tables. The most common relationships are based on equivalence. In this section, we examine the concept of an equi join.

Implementing two table joins with a table-dot notation

To see an example of the kind of join that *would* be advantageous to an SQL programmer, let's return to the earlier request to display name, date of birth, and address information for all of the *Companylink* employees. In the earlier examples in the chapter, we used two different queries to find the required information and noted the inefficiency of the process. To get this information using a join, we issue the query shown in the following screenshot:



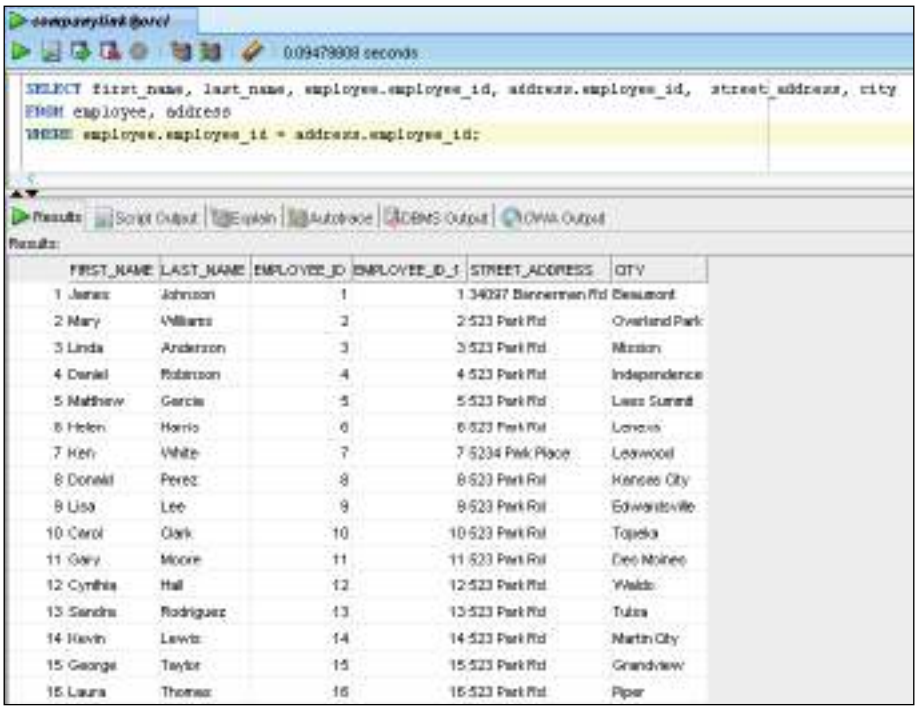
```
SELECT first_name, middle_initial, last_name, dob, street_address, city, state, zip
FROM employee, address
WHERE employee.employee_id = address.employee_id;
```

	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	DOB	STREET_ADDRESS	CITY	STATE	ZIP
1	James	R	Johnson	01-JAN-60	34097 Bannerman Rd	Beaumont	MN	23483
2	Mary	S	Williams	15-MAR-64	523 Park Rd	Overland Park	KS	44382
3	Linda	L	Anderson	24-OCT-76	523 Park Rd	Mission	KS	79529
4	Daniel	J	Robinson	25-NOV-58	523 Park Rd	Independence	MO	12335
5	Matthew	K	Garcia	14-APR-71	523 Park Rd	Lees Summit	MO	20772
6	Helen	H	Harris	13-JUL-75	523 Park Rd	Lenexa	KS	18632
7	Ken	W	White	22-FEB-56	5234 Park Place	Leawood	KS	78659
8	Donald	A	Perez	14-MAR-78	523 Park Rd	Kansas City	MO	12998
9	Lisa	C	Lee	15-JUN-63	523 Park Rd	Edwardsville	MO	36870
10	Carol	M	Clark	11-AUG-67	523 Park Rd	Topeka	KS	57819
11	Gary	R	Moore	01-NOV-65	523 Park Rd	Des Moines	IA	99272
12	Cynthia	B	Hall	21-OCT-55	523 Park Rd	Waldo	MO	45759
13	Sandra	S	Rodriguez	10-MAY-74	523 Park Rd	Tulsa	OK	38940
14	Kevin	L	Lewis	01-JUL-76	523 Park Rd	Martin City	MO	32736
15	George	H	Taylor	24-DEC-72	523 Park Rd	Grandview	MO	61056
16	Laura	I	Thomas	26-OCT-81	523 Park Rd	Piper	KS	28171

Let's deconstruct the statement one line at a time. The first line is simply a list of all the columns that we want to display. The first four are columns from the `employee` table and the last four come from the `address` table. The second line specifies the tables that are involved in the query. We are requesting columns from the `employee` and `address` tables, so those tables are listed. The third line contains the clause

that actually performs the join. We have stated that a join requires the linking of a common column between the tables. The only column that is common between the `employee` and `address` tables is the `employee_id` column; this column forms the relationship between the two tables. It is this column that we use to execute the join. The join clause performs this joining by creating a condition that sets the values for the `employee_id` column in the `employee` table equal to the `employee_id` column in the `address` table. Thus, each row in the `address` table is joined to each row in the `employee` table, but only where the values in the common columns are equivalent. The clause, however, requires that we specify which columns are being referred to, since they have the same name. To clearly delineate them, we prefix each of the columns in the `WHERE` clause with the name of the originating table, followed by a dot (`.`). We refer to this as the **table-dot notation**. As a result, the previous statement could be read as: *Display name and address information from the employee and address tables, where the employee_id column in the employee table is equivalent to the employee_id column in the address table*.

To see the relationship more clearly, we could rewrite the preceding statement to include the columns that form the relationship. In the following example, we have reduced the number of columns returned (for clarity) and included both columns that form the relationship. The result shows how the `employee_id` values for each table match.



As you can see, the rows displayed essentially show two tables joined together. The `first_name`, `last_name`, and `employee.employee_id` columns all belong to the `employee` table. The `address.employee_id`, `street_address`, and `city` columns belong to the address table. Yet information from both tables can be displayed together, provided that we join them with a common column. These joins are categorized as **equi joins**; joins based on the equivalence of values between common columns.

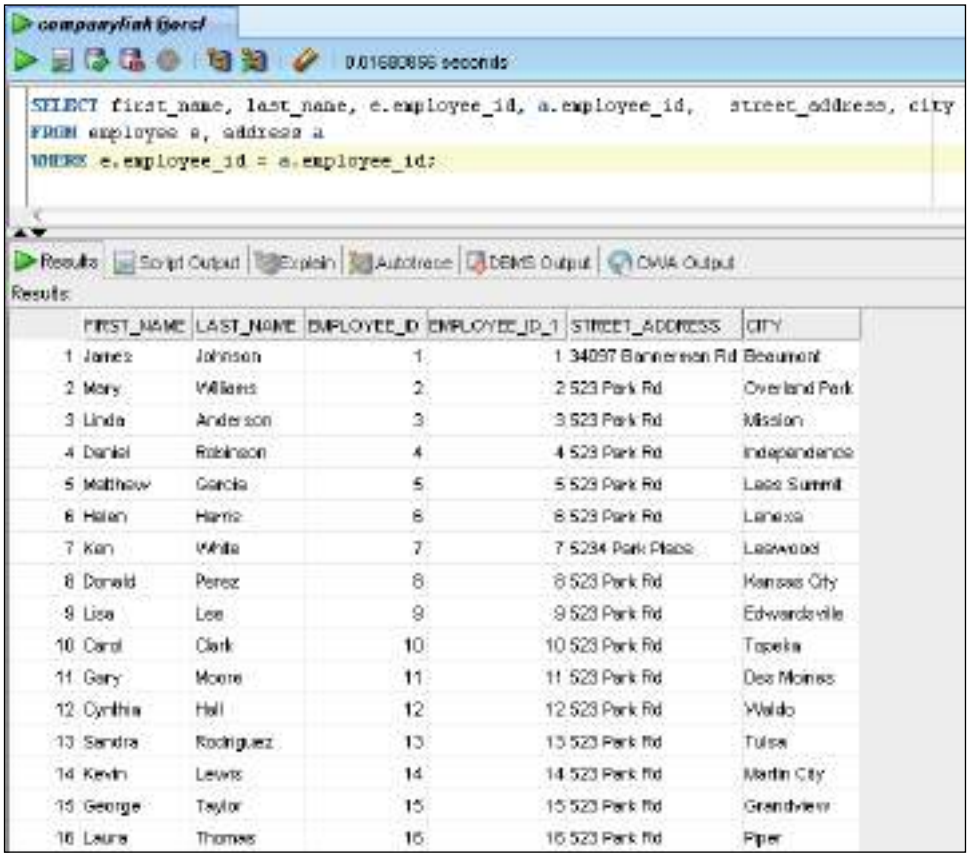
It is crucial that we explicitly define the two columns that form the join in the `WHERE` clause. A failure to do so will generate an **ORA-00918** error or a **column ambiguously defined** error.

The following example shows the previous statements rewritten to exclude the table-dot notation with the originating tables, and the resulting error. Again, this results because the columns have the same name in each table, yet we have not defined the originating tables.

Using two table joins with alias notation

Thus far, our join examples have used two tables. However, a join can be done with any number of tables, provided they have common columns between them. We will discuss multi-table joins later in the chapter, but using the table-dot notation with multi-table joins is considered by some to be cumbersome, since each table and several columns must be prefixed with the associated table name. As an alternative to table-dot notation, many SQL coders use **alias notation**.

We examined one form of alias in *Chapter 2*, *SQL SELECT Statements*. With that type, we used double quotation marks to present column headings that are different than the actual column name. It allowed us not only to display a different column name, but also to utilize case sensitivity and whitespace. This type of alias is called a **column alias**. We now look at a **table alias**, a type of alias that allows us to reference a table using a different name. In the following screenshot, we have rewritten the join shown previously, this time using table aliases, or alias notation:



	FIRST_NAME	LAST_NAME	EMPLOYEE_ID	EMPLOYEE_ID_1	STREET_ADDRESS	CITY
1	James	Johnson	1	1	34097 Bannerman Rd	Beaumont
2	Mary	Williams	2	2	523 Park Rd	Overland Park
3	Linda	Anderson	3	3	523 Park Rd	Mission
4	Daniel	Robinson	4	4	523 Park Rd	Independence
5	Matthew	Garcia	5	5	523 Park Rd	Lees Summit
6	Helen	Harris	6	6	523 Park Rd	Lenexa
7	Ken	White	7	7	5294 Park Place	Leawood
8	Donald	Perez	8	8	523 Park Rd	Kansas City
9	Lisa	Lee	9	9	523 Park Rd	Edwardsville
10	Carol	Clark	10	10	523 Park Rd	Topeka
11	Gary	Moore	11	11	523 Park Rd	Des Moines
12	Cynthia	Hall	12	12	523 Park Rd	Waldo
13	Sandra	Rodriguez	13	13	523 Park Rd	Tulsa
14	Kevin	Lewis	14	14	523 Park Rd	Martin City
15	George	Taylor	15	15	523 Park Rd	Grandview
16	Laura	Thomas	16	16	523 Park Rd	Piper

This example is very similar to the original statement, but there is one significant difference. In the `FROM` clause, we have added aliases for our two tables. These aliases are designated by the letters that follow each of the table names. For the `employee` table, the alias is `e`, and for the `address` table, it is `a`. Thus, in the `WHERE` clause, instead of prefixing the table names `employee` and `address` to our columns, we simply use `e` and `a` in place of them. These same aliases are also used in the `SELECT` clause, where `employee.employee_id` and `address.employee_id` are simply written as `e.employee_id` and `a.employee_id`, respectively. Note that there is nothing particularly significant about using the letters `a` and `e` as aliases. We could just as easily have used *emp* and *addr* as our aliases. The purpose is simply to reduce the amount of coding that has to be written. Many SQL developers feel that using aliases is an efficient and readable way to write joins, especially those that involve numerous tables.

SQL in the real world

Although both table-dot notation and alias notation are supported in ANSI-compliant joins, the organization you work for may decide that one is preferable to the other. As with many of the choices offered to a SQL programmer, an organization's coding standards may determine how code will be written. This isn't to say that one way is necessarily better than the other, but rather to support the idea of having standards for the code written in an organization. Code standards provide rules for writing and reading code that generally lead to better interoperability between programmers.

Understanding the row inclusiveness of outer joins

The joins we've seen thus far obey one simple rule—they only produce a corresponding row if there is a match between the values of a column that is common to both tables. Joins that meet this one-for-one matching criteria are categorized as **inner joins**. There are, however, certain circumstances in which we may wish to include values that *do not* match. Examine the following screenshot:

companylink@orcl

0.14439601 seconds

```
SELECT first_name, last_name, project_name
FROM employee e, project p
WHERE e.project_id = p.project_id;
```

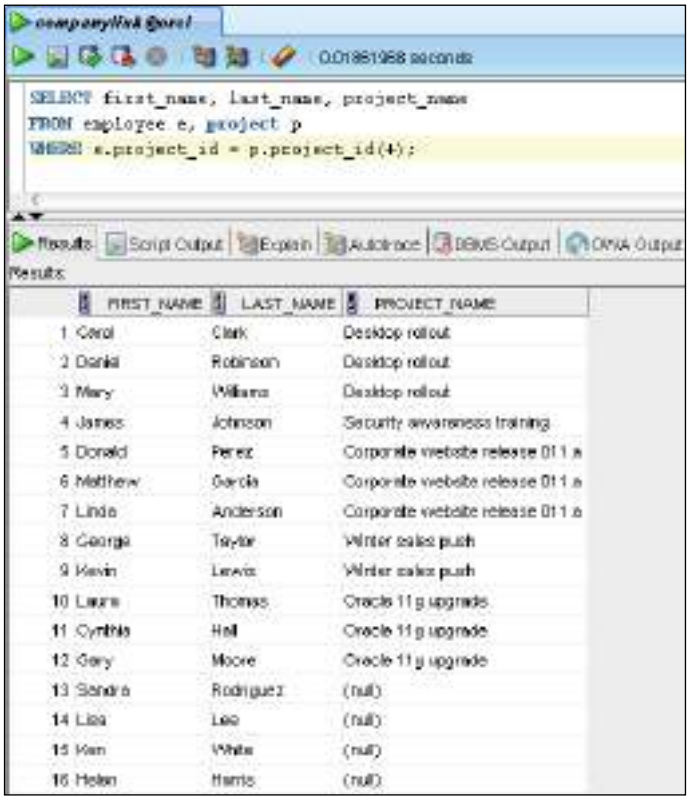
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	FIRST_NAME	LAST_NAME	PROJECT_NAME
1	James	Johnson	Security awareness training
2	Mary	Williams	Desktop rollout
3	Linda	Anderson	Corporate website release 011.a
4	Daniel	Robinson	Desktop rollout
5	Matthew	Garcia	Corporate website release 011.a
6	Donald	Perez	Corporate website release 011.a
7	Carol	Clark	Desktop rollout
8	Gary	Moore	Oracle 11g upgrade
9	Cynthia	Hall	Oracle 11g upgrade
10	Kevin	Lewis	Winter sales push
11	George	Taylor	Winter sales push
12	Laura	Thomas	Oracle 11g upgrade

Look carefully at the results of this query. In real-world terms, we have requested the names of each employee and the project to which they are assigned. However, our query has returned rows for 12 employees, while our `Companylink` database contains rows for 16 employees. This means that the project name information for four employees was not included in the record set. Why would this be? This lack of inclusiveness occurs because not all employees have a value for the `project_id` column in the `employee` table. Instead, four of the employees have a null value in the `project_id` column. If you wish to see this, simply enter a `select * from employee;` query in SQL Developer. In essence, not all employees have been assigned to a project, thus their rows are not returned because there is no match of values.

However, if we wished to include these, for example, for a report that was inclusive of all employees, we could use an outer join. An **outer join** is a join that purposely includes null values along with matching values in the result set. To accomplish an outer join using ANSI-compliant syntax, we use a plus symbol within parentheses, or **(+)**, as shown in the following example:



As we can see, the four employees who were excluded from the original join are now included, along with null values for their `project_name`. Outer joins can be signified by the terms right, left, or full. In the ANSI syntax for outer joins, which term we use is determined by the placement of the `(+)` symbol. This example is known as a **left outer join**, since the **(+)** symbol for inclusiveness is placed on the *right* side of the query condition.

The inclusiveness of outer joins is often required in data reporting. It prevents values from "falling through the cracks". A report using the query shown in the example could be used to point out that some employees have not yet been assigned to a project; the query shown in the example previous to it lacks this information.

Notice that the query that we just saw is identical to the original query, except for the addition of the **(+)** symbol next to the `p.project_id` column. This symbol simply instructs Oracle to include any rows associated with null values found in the `project_id` column.

When potential SQL programmers are learning to do outer joins, a common syntactical question raised is, "which side do I put the (+) on?". This refers to the question of where the `(+)` symbol is placed in the statement. Do we put the `(+)` with the `e.project_id` column or with the `p.project_id` column? This is an important question, since the placement of the `(+)` symbol in an outer join is critical. Fortunately, there is an easy rule of thumb to use in order to remember this. *Always place the (+) symbol with the table that does not contain null values*. In this case, that would be the project table. For instance, the `project_id` column in the `employee` table, or `e.project_id`, includes null values. The `project_id` column in the `project` table, `p.project_id`, contains no nulls. In order to properly execute an outer join of the two tables, we place the `(+)` symbol with the table that has no nulls, the `project` table and the `p.project_id` column. Since the `project` table contains all the values for `project_id` and the `employee` table lacks some of these values, we place the `(+)` with the column from the project table, `p.project_id`. In the following example, we show an example of improper placement of the `(+)` symbol. Here, we execute the same query as the original, correct outer join, but we place the `(+)` with the `e.project_id` column from the `employee` table. The `(+)` is paired with the wrong table. The resulting record set is only 12 rows instead of the inclusive 16 we were attempting to retrieve, since we have not properly instructed Oracle to retrieve rows with a null value. Always be aware of the placement of the `(+)` symbol in outer joins.

The following example demonstrates a **right outer join**, since the **(+)** symbol is on the *left* side of the condition.

companylink@orcl

0.01359167 seconds

```
SELECT first_name, last_name, project_name
FROM employee e, project p
WHERE e.project_id(+) = p.project_id;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

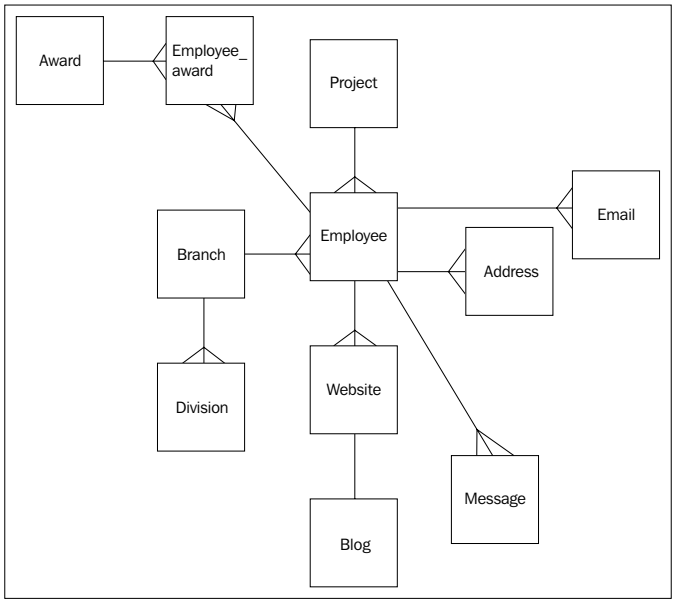
	FIRST_NAME	LAST_NAME	PROJECT_NAME
1	James	Johnson	Security awareness training
2	Mary	Williams	Desktop rollout
3	Linda	Anderson	Corporate website release 011.a
4	Daniel	Robinson	Desktop rollout
5	Matthew	Garcia	Corporate website release 011.a
6	Donald	Perez	Corporate website release 011.a
7	Carol	Clark	Desktop rollout
8	Gary	Moore	Oracle 11g upgrade
9	Cynthia	Hall	Oracle 11g upgrade
10	Kevin	Lewis	Winter sales push
11	George	Taylor	Winter sales push
12	Laura	Thomas	Oracle 11g upgrade

Outer joins have three configurations—right, left, and full. We have seen examples of both right and left outer joins. In a **full outer join**, the `(+)` symbol is placed with *both* columns, allowing the join to include null values from either column in the resulting set of rows.

In the Oracle proprietary join syntax, outer joins are much easier to interpret. Right, left, and full outer joins are constructed using the `RIGHT OUTER JOIN`, `LEFT OUTER JOIN`, and `FULL OUTER JOIN` keywords, respectively. The `(+)` symbol is not used.

Retrieving data from multiple tables using n-1 join conditions

As we have seen, joins can be a very effective way to retrieve information from two different tables. However, situations can arise that require data to be retrieved from more than two tables. In these circumstances, we can use our previously-discussed join techniques to retrieve data from as many tables as we wish, provided that we can establish the proper relationships between them. For instance, say that we've been tasked with writing the SQL for a report that will retrieve the information for employees have created websites and blogs in *Companylink*. When we are asked to create joins over multiple tables, it is often advantageous to have an **ERD**, or **Entity Relationship Diagram**, to which to refer. We introduced ERDs in *Chapter 1, SQL and Relational Databases*. They are used to graphically represent the relationships between tables. Let's look at an ERD for our `Companylink` database. It is shown in the following diagram:



Study this diagram closely. The tables are each represented by their names inside of boxes. The lines between the tables indicate the type of relationship between them. Tables can have different kinds of relationships, although the most common are *one-to-one* and *one-to-many*. In a one-to-one relationship, there is a "one and only one" relation between each of the rows in the tables. An example of this is the relationship between the `website` table and the `blog` table, shown by a single solid line between the two. This indicates that for each row in the `blog` table, there is one and only one corresponding row in the `website` table. Thus, we can locate the common column in the two tables— `blog_id`—and join the two tables on that column.

The more prevalent type of relationship between tables in a relational model is the one-to-many relationship. We can see an example of this between the `employee` and `email` tables, denoted by the line and "crow's foot" (three diverging lines) terminating at the table. When we see this diagram we can interpret it as: for every row in the `employee` table, there can exist one or more corresponding rows in the `email` table. In simpler terms, we can say that based on this diagram, each employee can have one or more e-mail addresses. Using an ERD can be a powerful, efficient way to form our joins.

Let us return to the requirement given to us at the beginning of this section, that is, to write an SQL query for a report that will retrieve the information for employees who have created websites and blogs in Companylink. We can surmise that we will need to select data from the `employee`, `website`, and `blog` tables: a situation that will require a join. From our ERD, we can see that although relationships exist between `employee` and `website`, and `website` and `blog`, no direct relationship exists between the `employee` and `blog` tables. How, then, can we retrieve the requested data? Even though no direct relationship exists between the `employee` and `blog` tables, an indirect relationship does. This indirect relationship exists because **employee** relates to **website** and **website** relates to **blog**. In essence, we can *get from* `employee` *to* `blog` through the `website` table. Doing so will require a multi-table join, as shown in the following screenshot:

companylink@orcl

0.04682327 seconds

```
SELECT first_name, last_name, website_desc, blog_desc
FROM employee e, website w, blog b
WHERE e.employee_id = w.employee_id
AND w.blog_id = b.blog_id;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

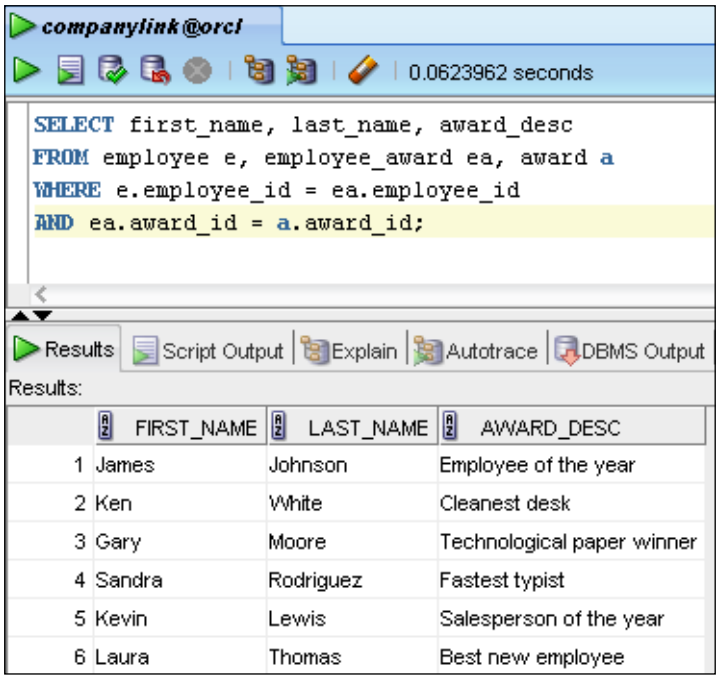
	FIRST_NAME	LAST_NAME	WEBSITE_DESC	BLOG_DESC
1	James	Johnson	Jims new site	Jims blog
2	Mary	Williams	Desktop rollout project site	Desktop rollout progress blog
3	Matthew	Garcia	Matts cool website	Matts cool blog
4	Gary	Moore	Garyworld!	Garyblog!
5	Kevin	Lewis	Winter sales push project site	Winter sales push blog

As we can see, this multi-table join is similar to our two-table joins, with two exceptions. First, our `FROM` clause contains three tables (the three targets of our query) instead of two. These three table names are separated by commas and denoted with aliases. Second, there is an additional join condition specified. Our first condition, the joining of `employee` and `website`, based on the `employee_id` column, is specified after the `WHERE` clause. Our second condition, the joining of `website` and `blog`, is specified following a Boolean `AND` operator. In short, we only retrieve rows if both join conditions are true.

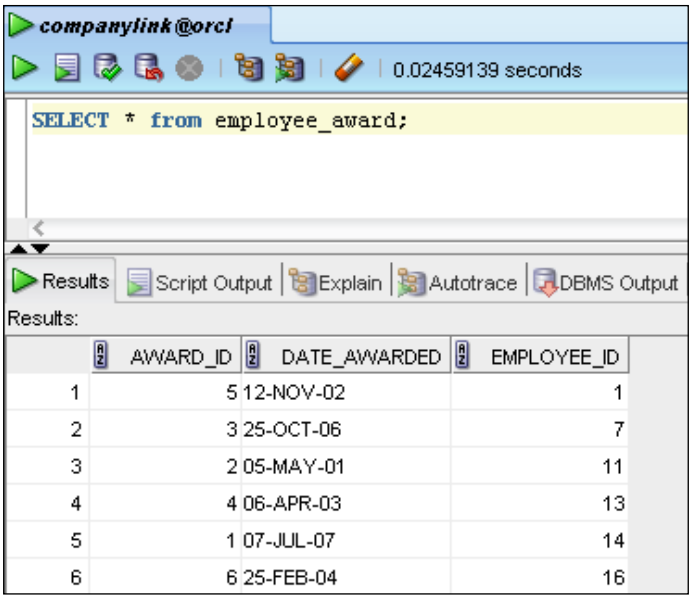
Notice that the number of conditions, or joins, that must be specified is one less than the number of tables involved. This is always the case.

Thus, we can say that *for any number of tables, n, that must be joined, there must exist n-1 join conditions*. We can refer to this as the n-1 join rule. It is true no matter how many tables are involved. Thus, if we need to join 20 tables, 19 join conditions will be required.

In our example, even though we used the website table to *get from* the `employee` table to the `blog` table, it is not required that we include any columns from the intermediary table in our query. For instance, let's say that we receive the requirement, *display all employees who have received awards and the name of the award they received*. To begin, we refer back to our ERD. We recognize that we will need data from the `employee` table and the `award` table, so we locate them on the diagram. We see, however, that they have no direct relationship. Even though no direct relationship exists, we notice that the `employee_award` table exists between the two. Thus, we can use n-1 join conditions (in this case, two conditions) to join three tables, as shown in the following screenshot. We could interpret this query as, *display all employees who have received awards and the name of the award they received*.



Here, even though we utilize the `employee_award` table as a *bridge* between the tables that contain the data we want, we do not retrieve any columns from it. We are only required to make use of it in order to form a relationship between the `employee` and `award` tables. In fact, the `employee_award` table is interesting because it holds very little actual `Companylink` data, as we can see from the following screenshot:



	AWARD_ID	DATE_AWARDED	EMPLOYEE_ID
1	5	12-NOV-02	1
2	3	25-OCT-06	7
3	2	05-MAY-01	11
4	4	06-APR-03	13
5	1	07-JUL-07	14
6	6	25-FEB-04	16

As we can see, beside the ID values from the `employee` and `award` tables, the `employee_award` table only contains one valuable column—`date_awarded`. If we were to attempt to form a relationship between *only* the `employee` and `award` tables, we would refer to this as a *many-to-many* relationship; meaning more than one employee can have more than one award. In the relational paradigm, this is generally considered unacceptable due to the problems of forming direct relationships. For our model to be fully relational, we must *resolve* this many-to-many relationship with a *bridging entity*. In our case, this bridging entity is the `employee_award` table, which has the primary function of completing the relationship between `employee` and `award`.

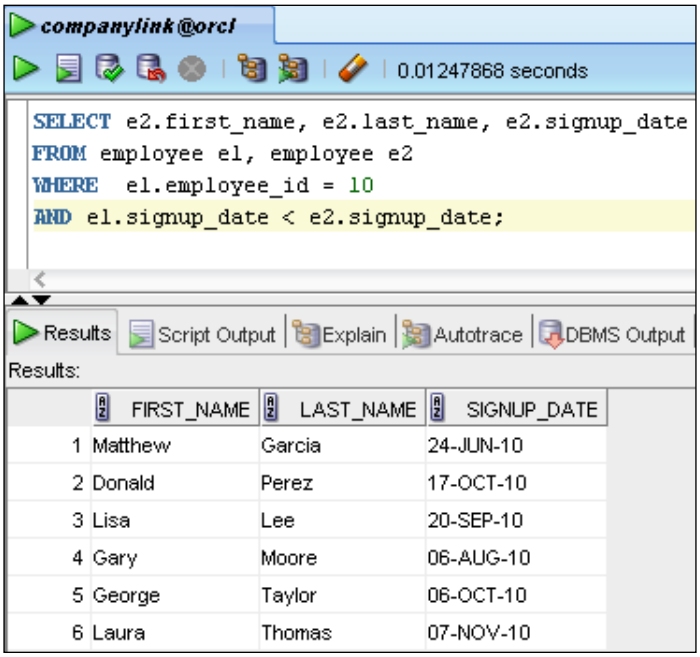
SQL in the real world

In real-world development situations, you may not always have an Entity Relationship Diagram at your disposal. Even though an ERD is tremendously useful, the lack of one should not prevent you from being able to construct complex joins. Instead, simply look at the tables with your target data and follow the trail of common values between them. It is often helpful to sketch the tables and columns out on a piece of paper and draw your own relationships.

Working with less commonly-used joins—non-equi joins and self-joins

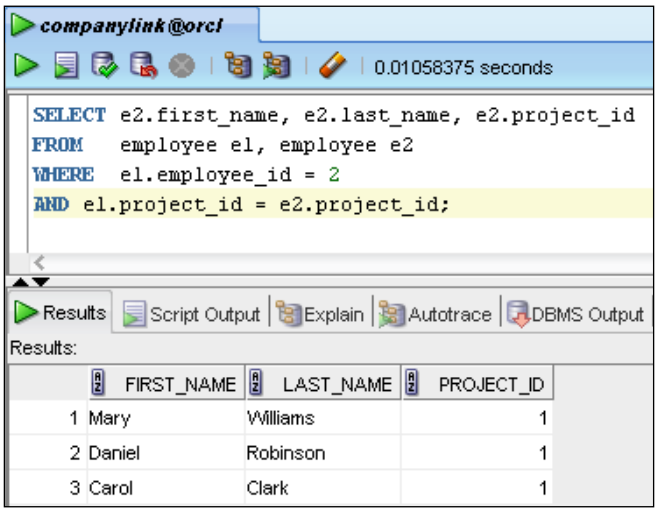
Our final look at ANSI-compliant joins will conclude with a brief discussion of two uncommon types of joins: non-equi joins and self-joins. These types of joins are less frequently used in actual development and are, sometimes, only considered necessary in an improperly constructed data model. Nevertheless, we review them here for the sake of completeness.

Although joins are typically constructed using equivalent values in common columns, which we have referred to as equi joins, we can join tables based on conditions of non-equality. A **non-equi join** is formed between tables where the join condition is based on any condition other than an equal sign, which can include `<`, `>`, `!=`, `<>`, and `BETWEEN`. An example of a non-equi join is shown in the following screenshot:



In this example, we have used a new technique—joining two instances of the same table. We can do this because we have separately aliased the `employee` table as `e1` and `e2`. The two instances of the `employee` table are joined together based on two conditions. The first condition specifies an `employee_id` of `10`, limiting the `e1` instance of the table to only that row. The second condition, the non-equi join, returns all values with a `signup_date` in the `e2` instance that is greater than the `signup_date` in instance `e1`. Since the `e1` instance is reduced to only rows where `employee_id` equals `10`, the query returns rows greater than that value. In real-world language, this query could be described as, *Return employee information where the employee signed up for Companylink after employee #10*. A non-equi join does not always have to use two instances of the same table to complete the join, as in our example. Some non-equi joins create a join between two tables without a common column utilizing the `BETWEEN` clause. However, the table data must be structured in a way that makes this possible.

A table can also be *joined back* on itself using a **self-join**. We've seen joins between different tables where a common column is used as the join condition. However, a self-join is constructed when one column of a table is joined to a column in the *same* table. Self-joins are sometimes used where a developer needs to iterate over a single table. An example of a self-join is shown in the following screenshot. Again, it uses a single table, aliased as two instances.



In this statement, the first instance of the `employee` table, `e1`, is limited by rows having an `employee_id` equal to `2`. This is then joined to the `e2` instance, where the `project_id` values are equivalent. In real language, this query could be stated as, *Display the employee and project ID information for all employees who have the same project ID number as employee* #2. A self-join can also be accomplished without aliasing the same table, provided that the table has a column that relates back to a different column in the same table. An example might be an `employee` table with a `supervisor_id` column that contains the `employee_id` of the supervisor. Since the supervisor is also an employee, the `supervisor_id` column could be related back to the `employee_id` column.

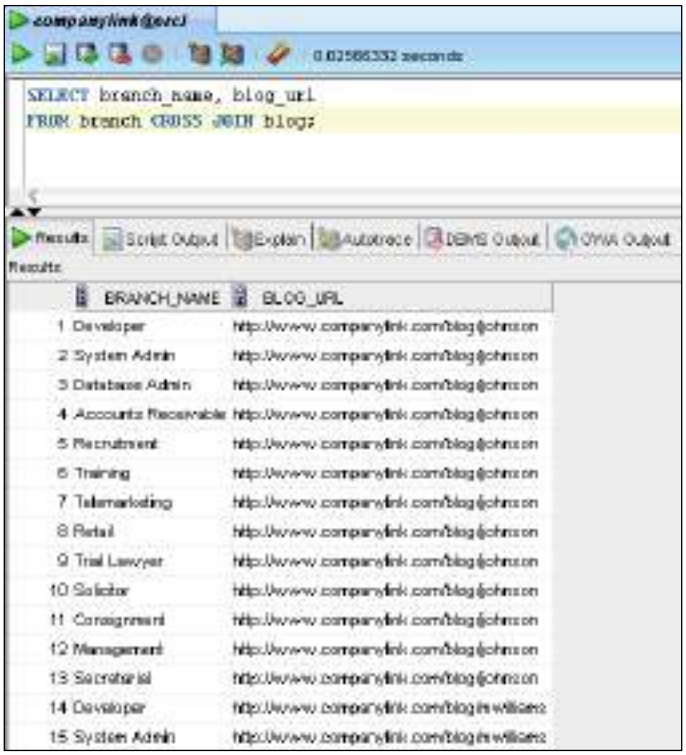
Understanding Oracle join syntax

Thus far, we've looked at the ANSI standard for the syntax used in joining tables. As noted previously, that standard is used throughout the information technology industry in many database management systems. However, when we are using Oracle, we have another choice—Oracle's new join syntax.

Beginning with database version 10g, Oracle has provided a new syntax for retrieving data from multiple tables. While Oracle's syntax has not yet gained widespread acceptance, we will see that it offers some advantages, not least of which is its simpler syntax. Many consider Oracle's join syntax to be more intuitive and readable than the ANSI standard.

Using Cartesian joins with Cross join

As we mentioned, the Cartesian product resulting from a Cartesian, or Cross join, is not always desirable. However, we can utilize it if necessary, using the Oracle `join` syntax as well. To do so, we use the `CROSS JOIN` clause, as shown in the following screenshot:



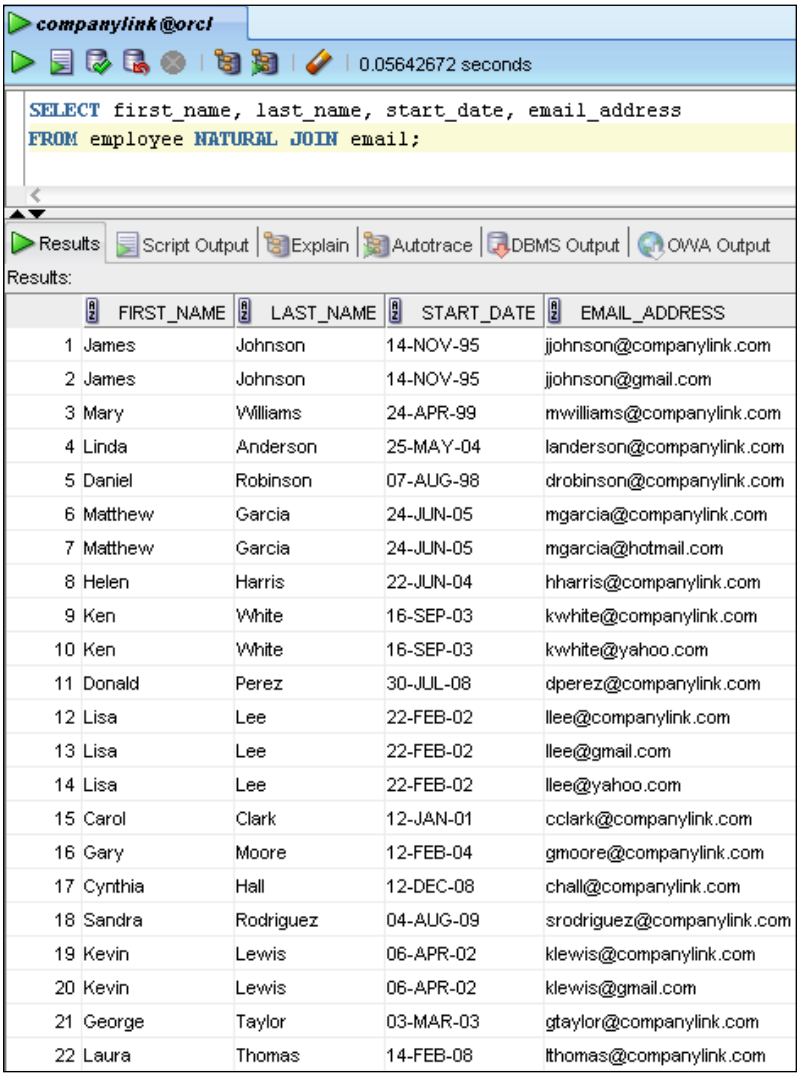
In this example, we select one column from each of the `branch` and `blog` tables— `branch_name` and `blog_url`, respectively. As we have noted with Cartesian joins, these tables have no relationship with each other; that is to say, they have no common columns.

The Oracle `CROSS JOIN` syntax is very similar to that of the ANSI-compliant join. The only real difference is the inclusion of the `CROSS JOIN` clause in place of a comma.

In the ANSI syntax, we make no distinction at all as to any relationship existing between the two tables. Thus, the ANSI syntax simply looks like a coding mistake, as if the coder simply forgot to add a `WHERE` clause. The Oracle syntax makes a purposeful inclusion of the `CROSS JOIN` clause to force the issue. With this clause included, we are explicitly stating that we are, in fact, attempting a Cartesian join between the two tables. Notice also that just as in the ANSI syntax, the resulting number of rows from the cross join is *a times b* rows, where *a* and *b* are the number of rows in the branch and blog tables, respectively. The `branch` table contains 13 rows and the blog table contains five rows, resulting in a 65 row result set.

Joining columns ambiguously using NATURAL JOIN

Since, as we've noted, cross joins rarely produce useful output, let's proceed by looking at one of the more useful clauses in the Oracle `join` syntax, the `NATURAL JOIN` clause. Let's say that we want to display employee name, starting date, and e-mail address information for a company e-mailer. To retrieve this information, we need to draw from two different tables. We can do this with a natural join, as shown in the following screenshot:



companylink@orcl

0.05642672 seconds

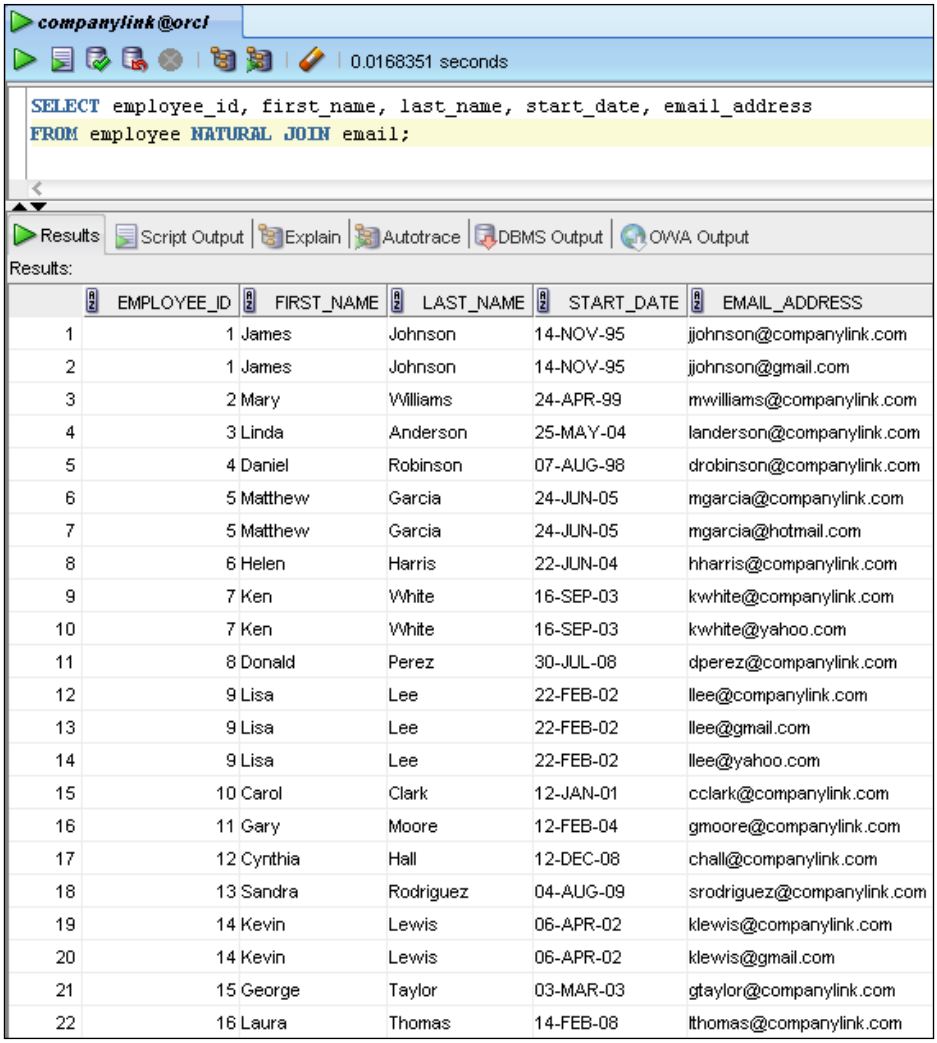
```
SELECT first_name, last_name, start_date, email_address
FROM employee NATURAL JOIN email;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	FIRST_NAME	LAST_NAME	START_DATE	EMAIL_ADDRESS
1	James	Johnson	14-NOV-95	jjohnson@companylink.com
2	James	Johnson	14-NOV-95	jjohnson@gmail.com
3	Mary	Williams	24-APR-99	mwilliams@companylink.com
4	Linda	Anderson	25-MAY-04	landerson@companylink.com
5	Daniel	Robinson	07-AUG-98	drobinson@companylink.com
6	Matthew	Garcia	24-JUN-05	mgarcia@companylink.com
7	Matthew	Garcia	24-JUN-05	mgarcia@hotmail.com
8	Helen	Harris	22-JUN-04	hharris@companylink.com
9	Ken	White	16-SEP-03	kwhite@companylink.com
10	Ken	White	16-SEP-03	kwhite@yahoo.com
11	Donald	Perez	30-JUL-08	dperez@companylink.com
12	Lisa	Lee	22-FEB-02	llee@companylink.com
13	Lisa	Lee	22-FEB-02	llee@gmail.com
14	Lisa	Lee	22-FEB-02	llee@yahoo.com
15	Carol	Clark	12-JAN-01	cclark@companylink.com
16	Gary	Moore	12-FEB-04	gmoore@companylink.com
17	Cynthia	Hall	12-DEC-08	chall@companylink.com
18	Sandra	Rodriguez	04-AUG-09	srodriguez@companylink.com
19	Kevin	Lewis	06-APR-02	klewis@companylink.com
20	Kevin	Lewis	06-APR-02	klewis@gmail.com
21	George	Taylor	03-MAR-03	gtaylor@companylink.com
22	Laura	Thomas	14-FEB-08	lthomas@companylink.com

The output from this statement gives us the desired information. We can also see that unlike in a cross join, the rows from each table are properly joined together. No extraneous rows are produced. The striking fact about this statement is that it contains no `WHERE` clause. In ANSI-compliant joins, we used a `WHERE` clause to set common column values equal. This would *instruct* Oracle as to how to complete the join. In the previous example, no common columns are specified. How, then, does Oracle know how to complete the join? Let's add to the complexity of the situation by adding the `employee_id` column to our statement, as shown in the following screenshot:



companylink@orcl

0.0168351 seconds

```
SELECT employee_id, first_name, last_name, start_date, email_address
FROM employee NATURAL JOIN email;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	START_DATE	EMAIL_ADDRESS
1	1	James	Johnson	14-NOV-95	jjohnson@companylink.com
2	1	James	Johnson	14-NOV-95	jjohnson@gmail.com
3	2	Mary	Williams	24-APR-99	mwilliams@companylink.com
4	3	Linda	Anderson	25-MAY-04	landerson@companylink.com
5	4	Daniel	Robinson	07-AUG-98	drobinson@companylink.com
6	5	Matthew	Garcia	24-JUN-05	mgarcia@companylink.com
7	5	Matthew	Garcia	24-JUN-05	mgarcia@hotmail.com
8	6	Helen	Harris	22-JUN-04	hharris@companylink.com
9	7	Ken	White	16-SEP-03	kwhite@companylink.com
10	7	Ken	White	16-SEP-03	kwhite@yahoo.com
11	8	Donald	Perez	30-JUL-08	dperez@companylink.com
12	9	Lisa	Lee	22-FEB-02	llee@companylink.com
13	9	Lisa	Lee	22-FEB-02	llee@gmail.com
14	9	Lisa	Lee	22-FEB-02	llee@yahoo.com
15	10	Carol	Clark	12-JAN-01	cclark@companylink.com
16	11	Gary	Moore	12-FEB-04	gmoore@companylink.com
17	12	Cynthia	Hall	12-DEC-08	chall@companylink.com
18	13	Sandra	Rodriguez	04-AUG-09	srodriguez@companylink.com
19	14	Kevin	Lewis	06-APR-02	klewis@companylink.com
20	14	Kevin	Lewis	06-APR-02	klewis@gmail.com
21	15	George	Taylor	03-MAR-03	gtaylor@companylink.com
22	16	Laura	Thomas	14-FEB-08	lthomas@companylink.com

If we examine the two tables, we can see that the common column between the `employee` and `email` tables is the `employee_id` column. We can see this, but how does Oracle know it? It knows because the `NATURAL JOIN` clause allows for ambiguity in column names. A natural join with Oracle's syntax is *smart* enough to be able to locate the common column between two tables, provided that one exists. When the statement is executed, Oracle recognizes the request for a natural join, examines the two tables and *sees* that there is a column, `employee_id`, with the same name in both tables. It then makes the assumption that the `employee_id` column is the target for your join and joins the tables appropriately. In a sense, we could say that a `NATURAL JOIN` is *less strict*, syntactically, than a similar join done with ANSI syntax.

Also notice that the first column we retrieve, `employee_id`, has no table definition. We have not explicitly noted whether we wish to display the `employee_id` column from the `employee` table or the `email` table. Were we to attempt a statement like this with an ANSI join, we would receive a *column ambiguously defined* error. But, again, since the `NATURAL JOIN` clause allows for column ambiguity, the statement retrieves the column as requested. In truth, with this syntax, Oracle makes the assumption that it actually does not matter which table the column comes from, since when the tables are joined, the values produced for the common column are actually the same. In that example, the values match up side by side. This is the essence of how a join works—by equivalently joining the values from common columns. Oracle uses this concept to allow for column ambiguity in natural joins.

What would happen if we attempted to use the `NATURAL JOIN` clause in a statement with two tables that did *not* have a common column? We see the results of such an attempt in the following example:

companylink@orcl

0.02063977 seconds

```
SELECT award_desc, project_name
FROM award NATURAL JOIN project;
```

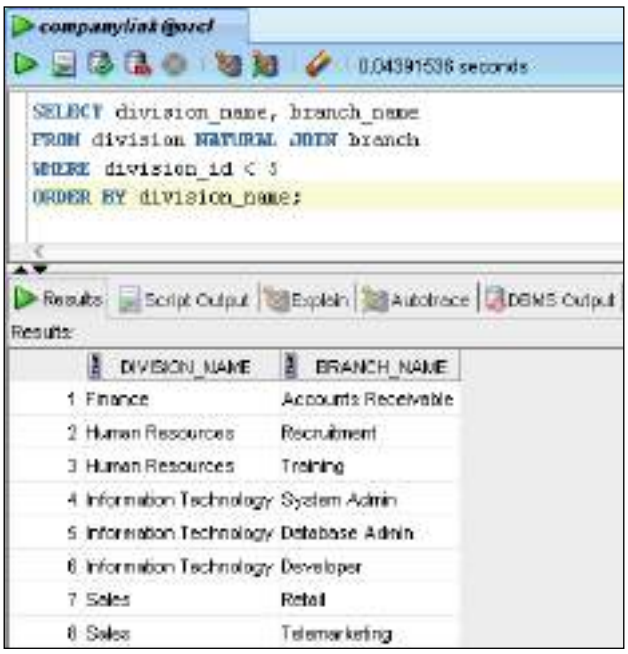
Results | Script Output | Explain | Autotrace | DBMS Output

Results:

	AWARD_DESC	PROJECT_NAME
1	Salesperson of the year	Desktop rollout
2	Technological paper winner	Desktop rollout
3	Cleanest desk	Desktop rollout
4	Fastest typist	Desktop rollout
5	Employee of the year	Desktop rollout
6	Best new employee	Desktop rollout
7	DML Guru	Desktop rollout
8	Salesperson of the year	Security awareness training
9	Technological paper winner	Security awareness training
10	Cleanest desk	Security awareness training
11	Fastest typist	Security awareness training
12	Employee of the year	Security awareness training
13	Best new employee	Security awareness training
14	DML Guru	Security awareness training
15	Salesperson of the year	Corporate website release 011.a
16	Technological paper winner	Corporate website release 011.a

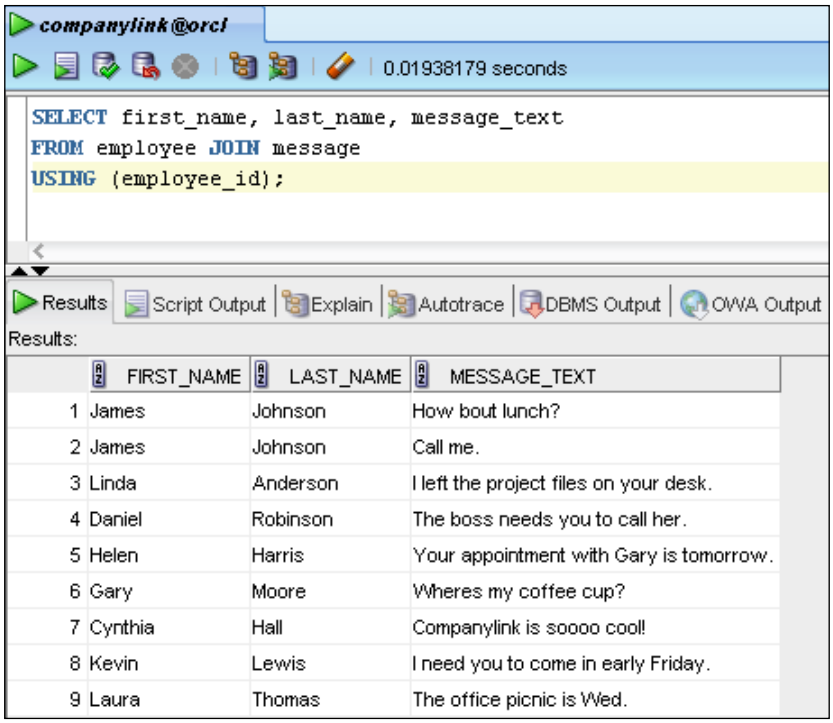
From the results, we see that a Cartesian product is formed. Oracle searches for a common column and, finding none, proceeds to join the two tables the only way it can—using a Cartesian, or Cross join. While it is true that Oracle's natural join syntax is less strict, this can lead to unforeseen problems unless proper care is taken to ensure that the natural join is constructed in such a way as to utilize a common column.

One of the benefits of using the Oracle `join` syntax is that it frees up the use of a `WHERE` clause. Since the syntax does not require the `WHERE` clause to establish equivalence between common columns, as in the case of the ANSI syntax, our statements can use the `WHERE` clause to its more common use—restricting row output. An example of this, that also includes a sort, is shown in the following screenshot. It retrieves the name of each division and its associated branch, but limits the output to only rows that have a **division_id** less than **5**. The output is then sorted alphabetically based on **division_name**.



Joining on explicit columns with JOIN USING

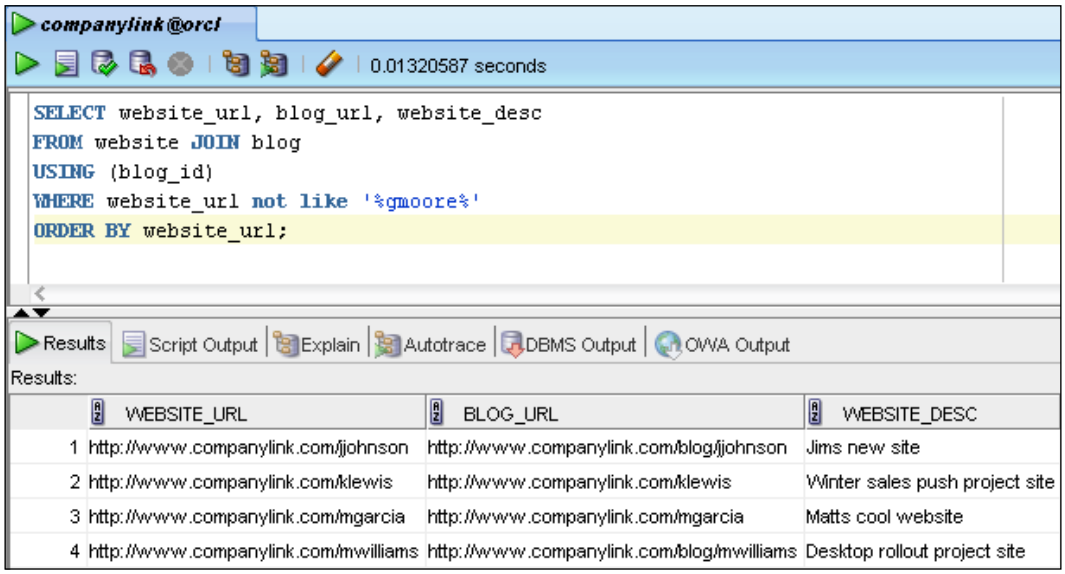
One of the criticisms of Oracle's `NATURAL JOIN` syntax stems from its ambiguity: a `NATURAL JOIN` allows columns to be joined without any specification as to what column will be used. While this allows for a more "natural" syntax, the ambiguity leads to the production of SQL code that lacks the specificity required in some coding standards. In short, if the column being joined is not explicitly stated, the code can be more difficult to interpret, often leading to a greater number of human errors. To combat this problem, the Oracle `join` syntax also includes the ability to perform a table join using a column that is explicitly defined. An example of this is shown as follows:



Although it differs somewhat from the `NATURAL JOIN` syntax, `JOIN USING` is similarly straightforward. The `FROM` clause specifies the tables to be joined, in our example **employee** and **message**, separated by a **JOIN** clause. It is then followed by the **USING** clause that specifies, in parentheses, the common column to use for the join. Thus, in this statement, we join the **employee** and **message** tables using the **employee_id** column.

Consider this example of a real-world requirement from our `Companylink` database: *Display all the URLs for websites and blogs in the Companylink system, along with the website description; however, do not display Gary Moore's site (gmoore), and sort the results by their website URL*. To begin, we examine the `website` and `blog` tables and find that they share a common column—`blog_id`. This *ties* the `blog` table to the `website` table, since any of the blogs on *Companylink* must first have a website associated with them. Next, we can exclude Gary Moore's site by adding a `WHERE` clause to restrict output. Finally, we use an `ORDER BY` to sort the output.

Our resulting statement is shown in the following example:



Recall from earlier chapters that we are using the `not like` operator to reject any rows that contain the string 'gmoore', or Gary Moore. Again, using the Oracle `join` syntax allows us to use the `WHERE` clause for row restriction, instead of setting common columns equal. Although both syntaxes will work, Oracle's syntax could be considered cleaner since it separates the functionalities of the join and the row restriction into different clauses.

Constructing fully-specified joins using JOIN ON

Our final type of join, using the Oracle syntax, removes all of the ambiguity that characterizes the two previous types. Statements that utilize the `JOIN ON` clause are very close syntactically to ANSI joins. The main difference, again, is that the `JOIN ON` clause, like the previous joins that use the Oracle syntax, do not require a `WHERE` clause in order to perform the join. An example of using `JOIN ON` is shown in the following screenshot:

companylink@orcl

0.03031279 seconds

```
SELECT first_name, last_name, start_date, branch_name
FROM employee JOIN branch
ON (employee.branch_id = branch.branch_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	FIRST_NAME	LAST_NAME	START_DATE	BRANCH_NAME
1	James	Johnson	14-NOV-95	Training
2	Mary	Williams	24-APR-99	System Admin
3	Linda	Anderson	25-MAY-04	Developer
4	Daniel	Robinson	07-AUG-98	System Admin
5	Matthew	Garcia	24-JUN-05	Database Admin
6	Helen	Harris	22-JUN-04	Solicitor
7	Ken	White	16-SEP-03	Accounts Receivable
8	Donald	Perez	30-JUL-08	Developer
9	Lisa	Lee	22-FEB-02	Recruitment
10	Carol	Clark	12-JAN-01	System Admin
11	Gary	Moore	12-FEB-04	Database Admin
12	Cynthia	Hall	12-DEC-08	Database Admin
13	Sandra	Rodriguez	04-AUG-09	Secretarial
14	Kevin	Lewis	06-APR-02	Retail
15	George	Taylor	03-MAR-03	Management
16	Laura	Thomas	14-FEB-08	Management

Notice the similarities to an ANSI-compliant join. In the `FROM` clause, both tables to be joined are specified. However, in the Oracle syntax we're using, we utilize the `JOIN` clause instead of simply separating the tables with a comma (`,`). Likewise, we specify the column to use for the join, `branch_id`, and denote this with table notation. However, instead of using a `WHERE` clause to do this, we can use the `ON` clause in conjunction with the `JOIN` clause, and surround the common columns with optional parentheses. The result is a join that still permits the `WHERE` clause to be used exclusively for row restriction.

Although the previous example uses table notation to specify our columns, the `JOIN ON` syntax fully supports the use of alias notation as well. An example of this is shown as follows. Here, we simply rewrite the previous example to use aliases.

companylink@orcl

0.01396602 seconds

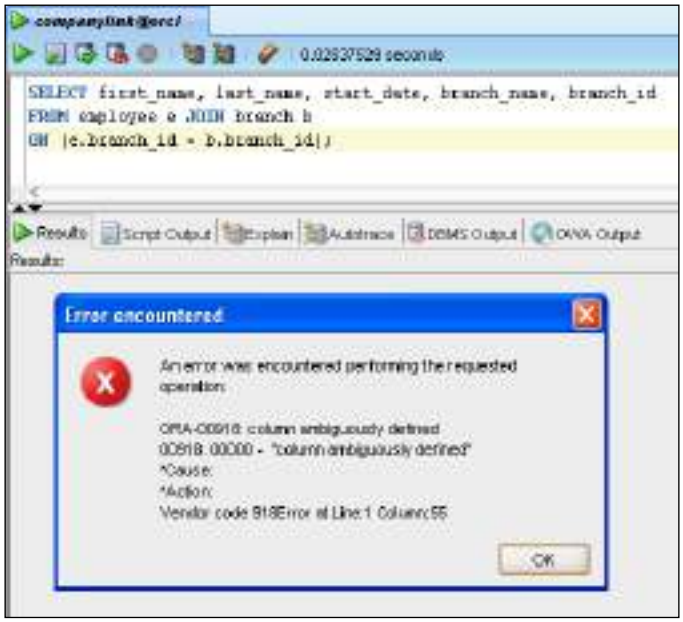
```
SELECT first_name, last_name, start_date, branch_name
FROM employee e JOIN branch b
ON (e.branch_id = b.branch_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	FIRST_NAME	LAST_NAME	START_DATE	BRANCH_NAME
1	James	Johnson	14-NOV-95	Training
2	Mary	Williams	24-APR-99	System Admin
3	Linda	Anderson	25-MAY-04	Developer
4	Daniel	Robinson	07-AUG-98	System Admin
5	Matthew	Garcia	24-JUN-05	Database Admin
6	Helen	Harris	22-JUN-04	Solicitor
7	Ken	White	16-SEP-03	Accounts Receivable
8	Donald	Perez	30-JUL-08	Developer
9	Lisa	Lee	22-FEB-02	Recruitment
10	Carol	Clark	12-JAN-01	System Admin
11	Gary	Moore	12-FEB-04	Database Admin
12	Cynthia	Hall	12-DEC-08	Database Admin
13	Sandra	Rodriguez	04-AUG-09	Secretarial
14	Kevin	Lewis	06-APR-02	Retail
15	George	Taylor	03-MAR-03	Management
16	Laura	Thomas	14-FEB-08	Management

If we wished to do so for clarity, we could prefix each of the columns in the `SELECT` statement with its appropriate alias to further clarify our code. Again, the `JOIN ON` syntax removes the ability to refer to columns ambiguously. If we modify our `SELECT` to include an unqualified column, `branch_id`, as shown in the next example, we receive the *ambiguous column* error that we've seen previously.

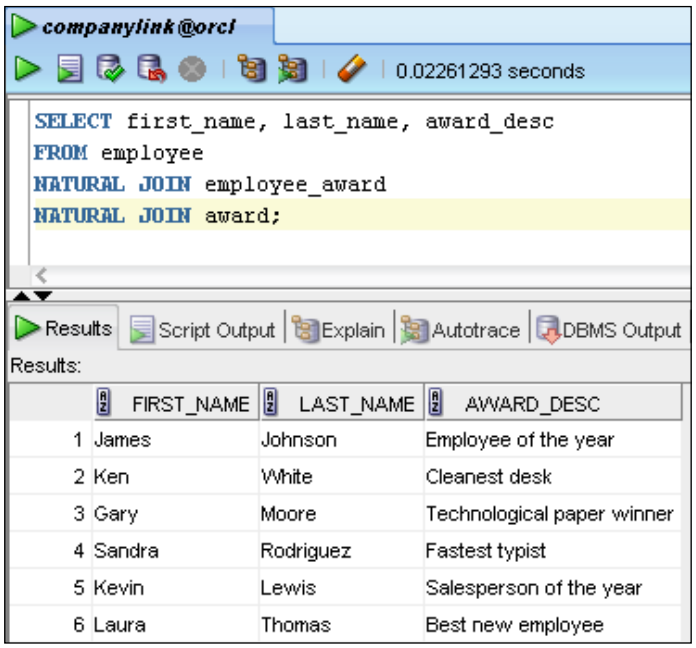


Writing n-1 join conditions using Oracle syntax

As with the ANSI syntax, multi-table joins using n-1 join conditions can be done with the Oracle syntax. We conclude the chapter by looking at two examples.

Creating multi-table natural joins

The syntax for using a natural join with multiple tables is similar to that of the two-table natural join we've seen previously. Again, it makes use of the `NATURAL JOIN` clause to ambiguously join tables based on common columns with the same name. An example is shown in the following screenshot:



In order to construct this join, we simply use the `NATURAL JOIN` syntax twice—once to join the `employee` table with `employee_award`, then a second time to join `employee_award` and `award`. Oracle locates the common columns between the tables and correctly performs the join. Again, this frees up the `WHERE` clause for additional restrictions on the rows displayed. Also note that the order of the `NATURAL JOIN` clauses is not important. Even if we attempt to natural join `employee` to `award` and then `award` to `employee_award`, the result is the same. Oracle is smart enough to parse the statement correctly.

Building multi-table joins with JOIN USING

For less ambiguity, we can also utilize the `JOIN USING` clause to specify the columns being used for the join. The syntax is similar to a two-table `JOIN ON`. We simply add additional clauses, as shown in the next example. Here, we join the three tables `employee`, `website`, and `blog` using two `JOIN USING` clauses.

companylink@orcl

0.01936559 seconds

```
SELECT first_name, last_name, website_url, blog_url
FROM employee
JOIN website USING (employee_id)
JOIN blog USING (blog_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	FIRST_NAME	LAST_NAME	WEBSITE_URL	BLOG_URL
1	James	Johnson	http://www.companylink.com/jjohnson	http://www.companylink.com/blog/jjohnson
2	Mary	Williams	http://www.companylink.com/mwilliams	http://www.companylink.com/blog/mwilliams
3	Matthew	Garcia	http://www.companylink.com/mgarcia	http://www.companylink.com/blog/mgarcia
4	Gary	Moore	http://www.companylink.com/gmoore	http://www.companylink.com/blog/gmoore
5	Kevin	Lewis	http://www.companylink.com/klewis	http://www.companylink.com/blog/klewis

SQL in the real world

As it stands in today's development world, your organization's coding standards may preclude the use of Oracle's relatively new join syntax. That does not mean it should be ignored. A good case can be made for the simplicity and readability of the Oracle syntax. Regardless, Oracle certification candidates must be extremely comfortable with the Oracle `join` syntax. It is covered extensively on the examination.

Summary

In this chapter, we've added the powerful capabilities of joins to our SQL repertoire. We've examined join techniques from two separate syntax families. We learned to write Cartesian joins, equi joins, non-equi joins, and self-joins in the ANSI syntax for both two-table and multi-table joins. With the Oracle syntax, we've looked at various join techniques using the `NATURAL JOIN`, `JOIN USING`, and `JOIN ON` clauses. We've also spent some time expanding our understanding of how table relationships work and graphically examined them for our `Companylink` database, in the form of an Entity Relationship Diagram, or ERD.

Certification objectives covered

- Write `SELECT` statements to access data from more than one table using equi joins and non-equi joins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using outer joins
- Generate a Cartesian product of all rows from two or more tables

In this chapter, we've learned various techniques in joining tables together. In our next chapter, we'll continue this idea of combining data together using a different technique—the subquery. Using subqueries, we can combine data from tables in new ways. We'll follow that subject up by looking at set operations and set theory in Oracle.

Test your knowledge

1. The relationships between tables are defined by what?
    a. Their query results
    b. The columns they have in common
    c. The number of rows they contain
    d. The number of columns they contain
2. In an ANSI-compliant join, which clause creates the equivalence relationship needed to form a join?
    a. The SELECT clause
    b. The FROM clause
    c. The WHERE clause
    d. The HAVING clause
3. Which of these is formed when every row of one table is joined to every row of another table?
    a. An intersecting product
    b. A union product
    c. A Cartesian product
    d. A truncated product

4. Which of these joins is generally considered the least useful in real-world situations?
   a. A Cartesian join
   b. An equi join
   c. A natural join
   d. A multi-table join

5. Given that two tables have 50 rows and 20 rows, respectively, how many rows would a Cartesian product of the two tables yield?
   a. 50
   b. 20
   c. 100
   d. 1000

6. Given two tables named employee and project, which of these WHERE clauses is a correct example of a join using the *table-dot* notation?
   a. WHERE e.project_id = p.project_id
   b. WHERE project_id = project_id
   c. WHERE employee_project_id = project_project_id
   d. WHERE employee.project_id = project.project_id

7. Given two tables named employee and project, which of these WHERE clauses is a correct example of a join using the *alias* notation?
   a. WHERE e.project_id = p.project_id
   b. WHERE project_id = project_id
   c. WHERE employee_project_id = project_project_id
   d. WHERE employee.project_id = project.project_id

8. Given the following SQL join statement, which line will cause an error?
   a. SELECT award_id, date_awarded
   b. FROM award a, employee_award ea
   c. WHERE a.award_id = ea.award_id;
   d. No error would be generated

9. What will be the result of the following SQL statement?

```
SELECT division_name, division_id, branch_name
FROM division div, branch brch
WHERE div.division_id = brch.division_id;
```

   a. A successful join with three columns and 13 rows
   b. A successful join with two columns and 13 rows
   c. A division by zero error
   d. An ORA-00918 error—column ambiguously defined

10. Which of the following is NOT a type of ANSI-compliant join?
   a. An inner join
   b. An outer join
   c. A natural join
   d. An equi join

11. Which symbol is used to declare an outer join?
   a. (*)
   b. (+)
   c. (-)
   d. %

12. If we are required to construct an equi join of 30 tables, how many join conditions will be required?
   a. 30
   b. 20
   c. 29
   d. The number cannot be determined

13. Which of these types of table relationships are not typically allowed in a relational model and should be resolved?
   a. One-to-one
   b. One-to-many
   c. Many-to-many
   d. None-to-none

14. Refer to the Companylink ERD displayed earlier in this chapter. Which of these pairs of tables cannot be related together, either directly or indirectly?
    a. Employee and award
    b. Branch and division
    c. Award and project
    d. None of the above

15. Which of the following clauses taken from an SQL statement would produce an error?
    a. FROM employee NATURAL JOIN award
    b. FROM employee NATURAL JOIN email
    c. FROM division d NATURAL JOIN branch b
    d. FROM employee JOIN award NATURAL

16. Which of the following clauses taken from an SQL statement is a syntactically correct example of Oracle's JOIN USING syntax?
    a. FROM employee, address JOIN USING (employee_id)
    b. FROM employee JOIN address USING employee_id
    c. FROM employee JOIN address USING (employee_id)
    d. FROM employee e JOIN address a USING (a.employee_id = e.employee_id)

17. Which of the following clauses taken from an SQL statement is a syntactically correct example of Oracle's JOIN ON syntax?
    a. FROM employee e JOIN address a ON (a.employee_id = e.employee_id)
    b. FROM employee JOIN address ON (employee_id = employee_id)
    c. FROM employee e JOIN ON address a (a.employee_id = e.employee_id)
    d. FROM employee e JOIN address a WHERE (a.employee_id = e.employee_id)

6
Row Level Data Transformation

Much of the work we have done so far with SQL has involved construction from scratch. We have examined different SQL clauses and investigated how they can be used together to accomplish a goal. But SQL has an entire class of built-in program units that can be used to greatly simplify our coding. These programs vary between the different implementations of SQL, but many commonalities exist between them. In this chapter, we look at how these built-in programs can enable us to manipulate data in ways that would be impossible with SQL clauses alone.

In this chapter, we shall:

- Examine the purpose of functions and how they work
- Learn what makes single-row functions unique
- Use string functions to transform character strings
- Work with arithmetic functions to transform numeric values
- Look at date functions and date arithmetic
- Examine functions that do conditional retrieval

Understanding functions and their use

One of the most fundamental constructs of any programming language is the function. Broadly speaking, a **function** is any unit of code that accomplishes a defined task. Functions are also referred to, depending on the language being used, as procedures, routines, or methods. In this section, we introduce the concept of a function and see how we can use them to transform database data.

Comprehending the principles of functions

A function allows a program to be structured into separate components, increasing the modularity, readability, and reusability of code. With functions, we are freed from the limitations of creating programs that simply execute one command after another in a purely linear fashion. Functions allow the grouping of code into discreet elements to accomplish a particular action. Functions adhere to different rules and definitions that vary between programming languages, so it is important to distinguish how the SQL language implements functions. In Oracle's implementation of SQL, a function simply takes the input that is passed to it and returns a value.

Using single-row functions for data transformation

For example, say that we are tasked with the following: display the first and last names of every employee in the `Companylink` database *in uppercase*. While we know from previous chapters that we could use a `SELECT` statement to display the first and last names from the `employee` table, how would we show them in uppercase? Doing so would probably require the use of additional code that could read each letter, determine its case, and decide to either change the ASCII value to that of the corresponding capital letter or leave the value alone. This would seem like a lot of extra work for something as common as **case conversion**. Fortunately, in situations like this, we can use Oracle's built-in functions.

The functions that we will see in this chapter and the one following it aid in **data transformation**—extracting data from the database and transforming it in some useful way. The ability to convert a lowercase string into uppercase is an example of data transformation, as is rounding a number, converting a date value into a string, or computing the average of several numbers. Many of the common tasks necessary in the day-to-day work of a SQL programmer involve data transformation.

SQL in the real world

Data transformation is a key component used in data warehouses, decision support systems, and online analytical processing environments. Oracle's built-in functions support the analysis and data mining efforts that enable systems such as these.

Oracle's built-in functions are provided with the RDBMS itself. They are included in Oracle's implementation of SQL. We have two types of functions at our disposal—single-row functions and multi-row functions.

- **Single-row functions** work by taking each row as an input value and returning a corresponding single value. An example of this would be a function to convert each value in the `first_name` column in the `employee` table to uppercase. When we execute the statement using a function, it processes each row at a time, taking in a value for `first_name` and returning the corresponding value in uppercase. If our table had 16 rows with `first_name`, 16 rows of uppercase values would be returned.
- **Multi-row functions** work by taking the values from multiple rows and returning only a single value. An example of a multi-row function would be one that averages the `login_count` column from the `employee` table. The multi-row function takes all of the values for `login_count` as input, but returns only one value; namely, the average of all values that were inputted. This chapter focuses on the many types of single-row functions available to us. Multi-row functions are the subject of the next chapter.

Understanding String functions

Our first examples of functions will examine many of the single-row functions that are capable of transforming string, or character, data. Of course, string data is one of the most common types of data stored in a database, so our ability to transform such data through the use of functions is critical. The types of string functions at our disposal include case conversion functions and string manipulation functions.

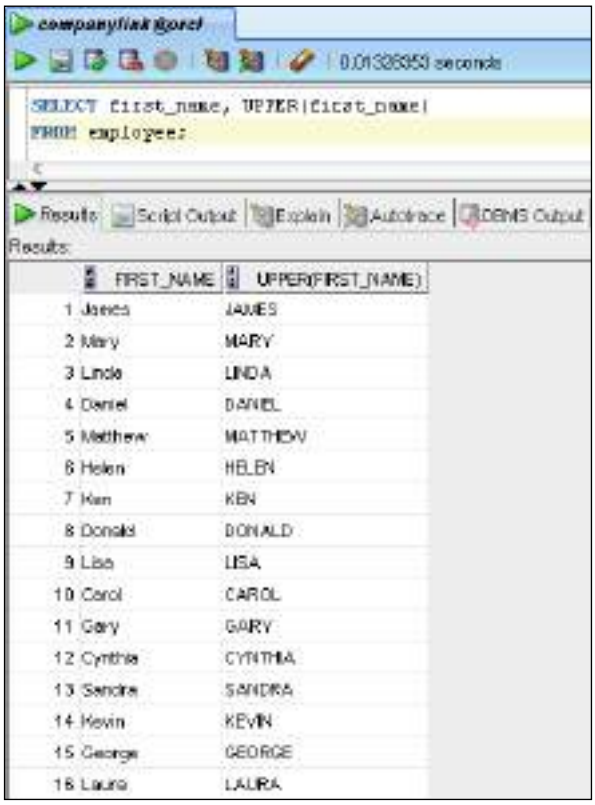
Using case conversion functions

Case conversion functions involve the transformation of strings from one case to another. These types of functions are extremely common, particularly in reporting. They include:

- `UPPER()`
- `LOWER()`
- `INITCAP()`

UPPER()

Our first case conversion function is `UPPER()`, which is used to convert a string into uppercase values. The original case of the string value is irrelevant; `UPPER()` will convert lower or mixed case values into uppercase. If the original string value is in uppercase, the resulting value will be unchanged, but no error will be generated. An example using `UPPER()` is shown in the following screenshot:

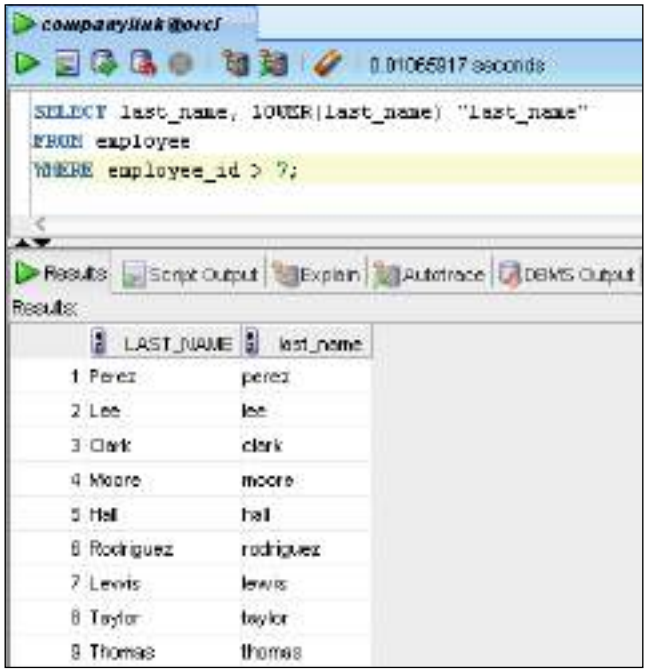


Our previous example is straightforward. We use a `SELECT` statement to display the `first_name` column from the `employee` table, followed by the same column with the function applied. In the second occurrence of `first_name`, we use the column as an argument to the `UPPER()` function. An **argument** is the value passed into the function. Different functions require different numbers and types of arguments. The `UPPER()` function both requires and allows only one argument—the value to convert. In our case, we pass the value for `first_name`, for each row, to the `UPPER()` function as an argument. Since `UPPER()` is a single-row function, one uppercase value is returned for each value of `first_name` taken as an argument. In short, we pass each value for `first_name` into the `UPPER()` function and return uppercase values.

Functions can be used multiple times within a single statement. Because the column heading includes the name of the function, it is sometimes clearer to use column aliases to improve the appearance of the resulting column headings. Both of these techniques are shown in the following example:

LOWER()

The `LOWER()` function performs the opposite operation of `UPPER()`—it takes the value from each row passed in as an argument and displays the string in lowercase, as shown in the following screenshot:

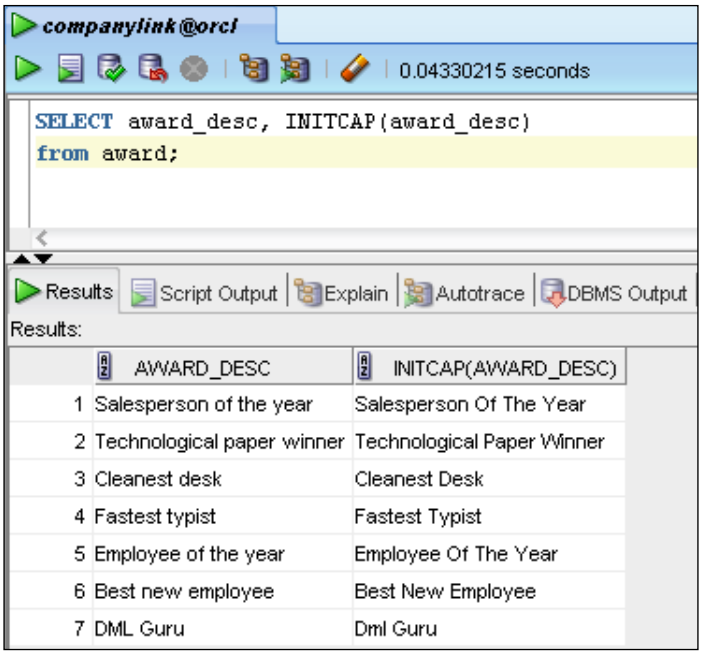


The unaltered values for `first_name` as they exist in the table and the corresponding transformed values are displayed. Keep in mind that, when we use functions with a `SELECT` statement, the original data in the table is unchanged. It is simply displayed in the way we have requested. Also notice that we have included a `WHERE` clause in our statement to reduce the number of rows returned. The use of functions does not prevent the use of clauses such as `WHERE` and `ORDERBY`.

INITCAP()

The `INITCAP()` function takes a column value as input and returns a value in mixed case. The case is mixed in such a way as to display it with the first letter of every word in capital letters. The following screenshot shows an example of this. Notice that the first row in the column without `INITCAP()` reads, **Salesperson of the year**. `Salesperson` is mixed case, while the remainder of the string value `of the year` is all lowercase. Once the `INITCAP()` function is applied, the resultant string `Salesperson Of The Year` contains individual words in mixed case. Notice, however, that the

`INITCAP()` can potentially introduce unwanted conversions, such as the one in the last row of the example. Here, **DML Guru** has been changed to **Dml Guru**. Since DML is a properly capitalized acronym, converting it to mixed case is probably an unwanted side effect.



	AWARD_DESC	INITCAP(AWARD_DESC)
1	Salesperson of the year	Salesperson Of The Year
2	Technological paper winner	Technological Paper Winner
3	Cleanest desk	Cleanest Desk
4	Fastest typist	Fastest Typist
5	Employee of the year	Employee Of The Year
6	Best new employee	Best New Employee
7	DML Guru	Dml Guru

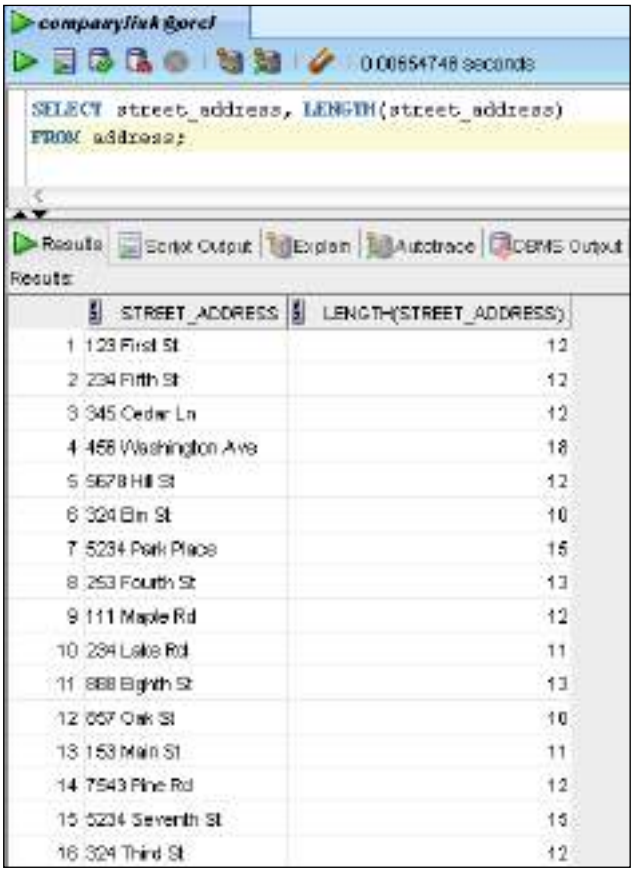
Writing SQL with String manipulation functions

Oracle contains many built-in functions whose purpose is to take inputted string data and return values that are meaningful in some way. **String manipulation functions** are used for this purpose. String manipulation functions can be used to locate character positions within strings, extract portions of data, and even replace values within a string. These types of functions are often used together, for instance, to locate a certain character within a string and then replace it with another.

LENGTH()

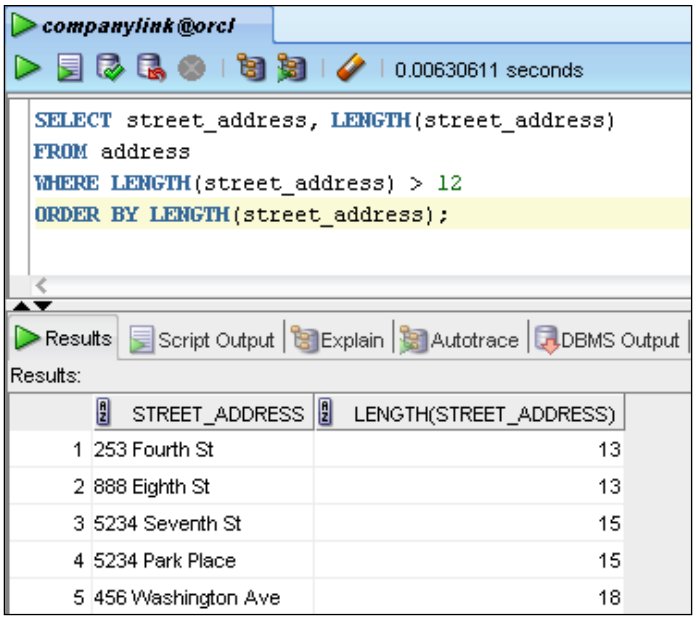
The SQL `LENGTH()` function takes a value as input and returns only the number of characters in the string. The original data itself is *not* returned. However, even though returning just the length of a string may seem inconsequential, this can be used for many purposes. Let's say developers are creating a web form to display employee address information. Part of this design is to know how to size the elements in the page in order to display the information properly. For instance, we don't want the section of the page that displays the `street_address` column to be smaller than the values in the table.

The `LENGTH()` function can aid us in finding the size of the values for the `street_address` column. Examine an example of this found in the following screenshot. We could interpret this statement as, *Display addresses and the number of characters in them*.



The output shows us that the length of each of the values for `street_address` is no less than `10` and no greater than `18`. With this information, developers can accurately size the web page elements to accommodate this.

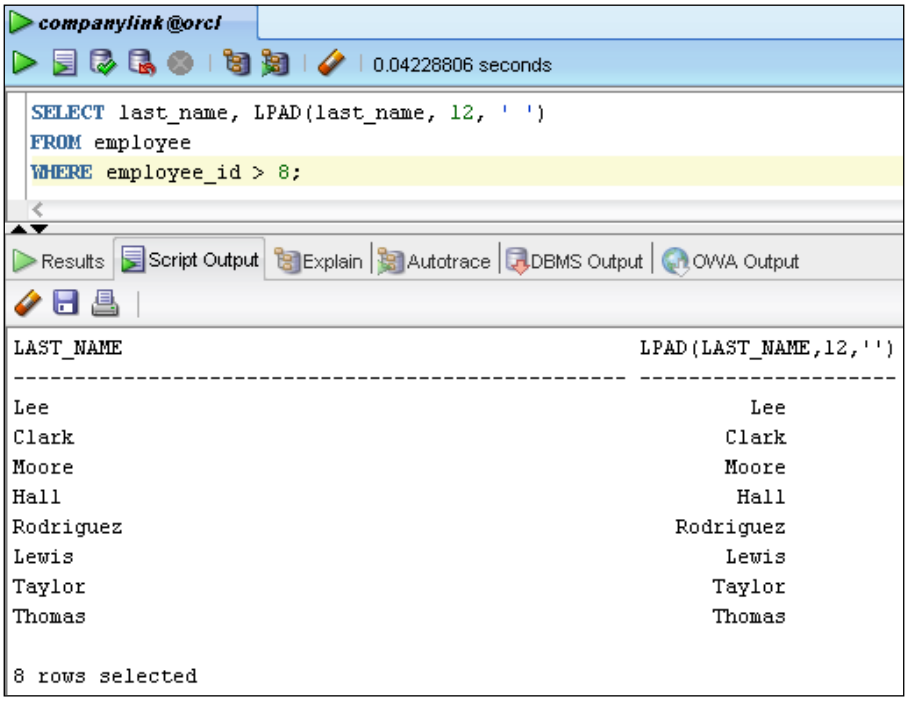
Although we have used functions only in conjunction with our `SELECT` clause, functions do not need to be limited in such a way. Another powerful way that functions can be used is with other clauses, such as `WHERE` and `ORDERBY`. For a realistic example, let's say that our developers have already sized the web page elements for address information. They have sized the box for `street_address` to accommodate addresses that are 12 characters in length. We want to know which street addresses will exceed that size. For this, we can incorporate the use of functions in the `WHERE` clause; that is, *where the length of the street address is more than 12 characters*. We can even sort the results by the length of the value. A solution for this request is shown in the following example. We could interpret this query as, *Which street addresses exceed the required length?*.



In this example, we display values for `street_address` and their corresponding lengths. We also limit this output to `street_address` values that are greater than `12` and order the results by the length of each value.

Padding characters with LPAD() and RPAD()

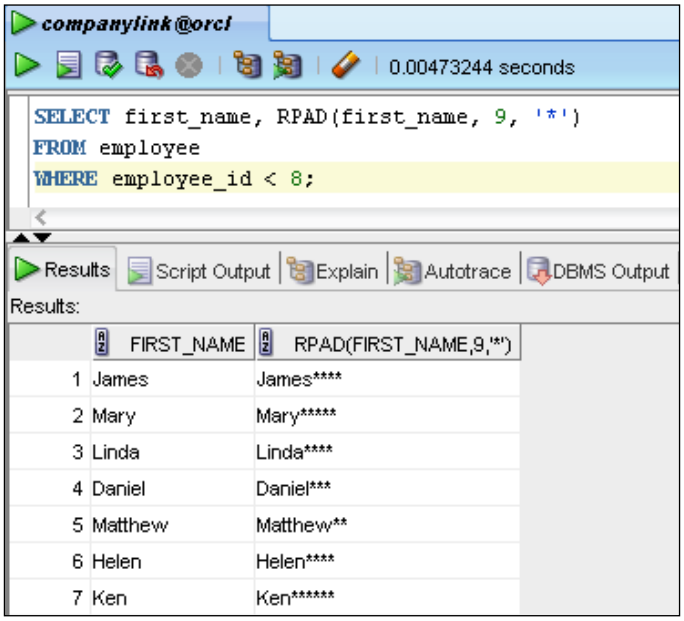
The padding functions, `LPAD()` and `RPAD()`, are used primarily for altering the way that data appears on the screen. These functions will add, or *pad*, characters to the left (`LPAD`) or right (`RPAD`) of the input values. The padding functions are our first examples of functions that take more than one argument as an input. Both of the padding functions take three arguments: the value to which to add padding, the total width of the value with padding, and the character with which to pad. An example of the `LPAD()` function is shown in the next screenshot. In order to best display the output, instead of executing the command with the **Execute Statement** button (the green arrow), we will once again use the **Run Script** button just to the right of the green arrow. Alternatively, you can press the *F5* key.



Let's look at the function portion of the statement more closely. For the first argument, we input **LAST_NAME**, which contains the values we wish to pad. The second argument, the integer **12**, is the total width of the value, plus the padding. The third argument we input is the actual character with which to pad; in our case, a space character, which must be enclosed in single ticks. Thus, the first value

returned, **Lee**, is three characters in length. Since we have inputted a value of 12 for the total width, the resulting value is **Lee** padded on the left with nine spaces, for a total width of 12. The second value, **Clark**, is five characters, so it is padded with seven spaces, for a total width of 12. This left padding has the effect of showing the values as *right-justified*.

Although the space character is commonly used for padding, any character or characters can be used. In the next query, we use the `RPAD()` function to pad asterisks to the right side of each value for **first_name** for a total width of **9**.



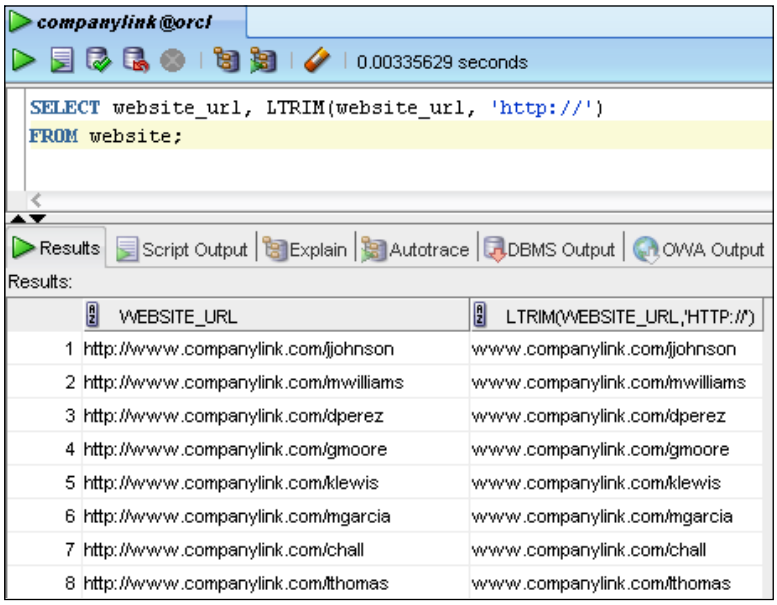
	FIRST_NAME	RPAD(FIRST_NAME,9,'*')
1	James	James****
2	Mary	Mary*****
3	Linda	Linda****
4	Daniel	Daniel***
5	Matthew	Matthew**
6	Helen	Helen****
7	Ken	Ken******

SQL in the real world

When using `LPAD()` and `RPAD()`, the resulting appearance of the data values is highly dependent upon the tool and font used to display the values. To see this for yourself, try running the previous example by hitting the **Execute Statement** button. It is much more difficult to see the padding in the values, since the font used to display them is not fixed width. Because of this, the padding functions are not used as commonly as they once were.

RTRIM() and LTRIM()

Just as values can be padded using functions, they can also be *trimmed* or removed from the left or right. To accomplish this, we use the `RTRIM()` function for trimming values from the right side and `LTRIM()` to trim them from the left. For example, examine the `website_url` column in the `website` table. In this column, we store the full URL for each user's website, including the `http://` prefix. If we wanted to display the URL without this prefix, we would need to *trim* the characters `'http://'` from the left side of each value. To do this, we can use the `LTRIM()` function as shown in the following query:



The trimming functions take two arguments; the value to trim and the string of characters we wish to remove from either the left (`LTRIM`) or right (`RTRIM`).

CONCAT()

Earlier in *Concatenating Values in SELECT Statements* section in *Chapter 2, SQL SELECT Statments*, we learned how to use the double-pipe symbol (`||`) to concatenate values together. At that time, we briefly mentioned that there were two ways to accomplish concatenation. The second method values can be concatenated by using the `CONCAT()` function, which accepts two arguments, the two values to be concatenated, and returns them appended together. The `CONCAT()` function is shown in our next example:

companylink@orcl

0.00800437 seconds

```
SELECT CONCAT(first_name, last_name)
FROM employee
WHERE employee_id < 9;
```

Results | Script Output | Explain | Autotrace | DBMS Output

Results:

	CONCAT(FIRST_NAME,LAST_NAME)
1	JamesJohnson
2	MaryWilliams
3	LindaAnderson
4	DanielRobinson
5	MatthewGarcia
6	HelenHarris
7	KenWhite
8	DonaldPerez

Although the `CONCAT()` function serves the same purpose as the double-pipe operator, it is limited in that it only takes two arguments. So, where the double-pipe operator can be used repeatedly to concatenate many values, the `CONCAT()` function can only append two values at a time.

SQL in the Real World

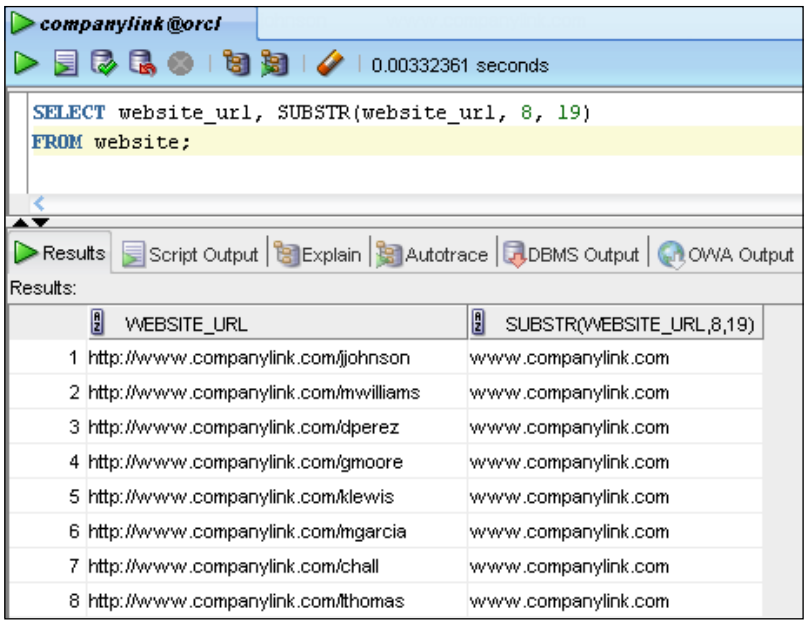
In order to concatenate multiple values together with `CONCAT()`, it would be necessary to nest multiple occurrences of the function inside each other. While this is possible, the resulting code can be difficult to read. For this reason, it is far more common to see the double-pipe used to concatenate values than the `CONCAT()` function.

SUBSTR()

One of the most powerful abilities of string manipulation functions is that of text extraction that allows us to remove sections of characters from string values. To do this, we can use the `SUBSTR()` function. The `SUBSTR()` function is somewhat more complex than the functions shown thus far, so we begin with an example syntax tree as shown here:

```
SELECT SUBSTR(column_expression, start position, end position)
FROM {table};
```

We see that the `SUBSTR()` function takes three arguments, although only the first two are strictly mandatory. The first argument, as we might expect, is the column itself. The second argument is the starting position to begin the substring. Thus, if the starting position was the number `4`, the statement would count over to the fourth character in the value and start the substring at that position. The third argument is the ending position, or end, of the substring, which begins counting from the start position. Say, for example, we want to pull the domain of each website from the URL listed in the website table. A statement using `SUBSTR()`, as shown in the following screenshot, can accomplish this:



	WEBSITE_URL	SUBSTR(WEBSITE_URL,8,19)
1	http://www.companylink.com/jjohnson	www.companylink.com
2	http://www.companylink.com/mwilliams	www.companylink.com
3	http://www.companylink.com/dperez	www.companylink.com
4	http://www.companylink.com/gmoore	www.companylink.com
5	http://www.companylink.com/klewis	www.companylink.com
6	http://www.companylink.com/mgarcia	www.companylink.com
7	http://www.companylink.com/chall	www.companylink.com
8	http://www.companylink.com/lthomas	www.companylink.com

Let's examine the first value returned to see how `SUBSTR()` works.

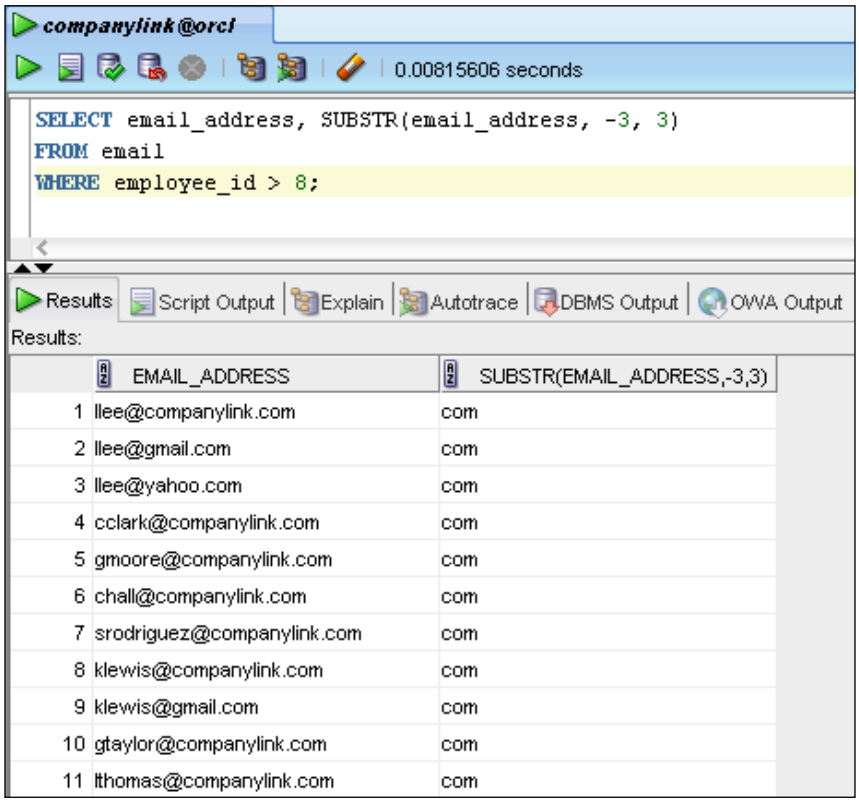
http://www.companylink.com/jjohnson

`SUBSTR()` begins with this value as the first argument. Next, it counts over to the eighth character, the first 'w', shown in bold, to designate as the beginning of the substring. Then, it counts over 19 more characters to the 'm', also shown in bold, to mark as the end of the substring. It then returns the following substring:

www.companylink.com

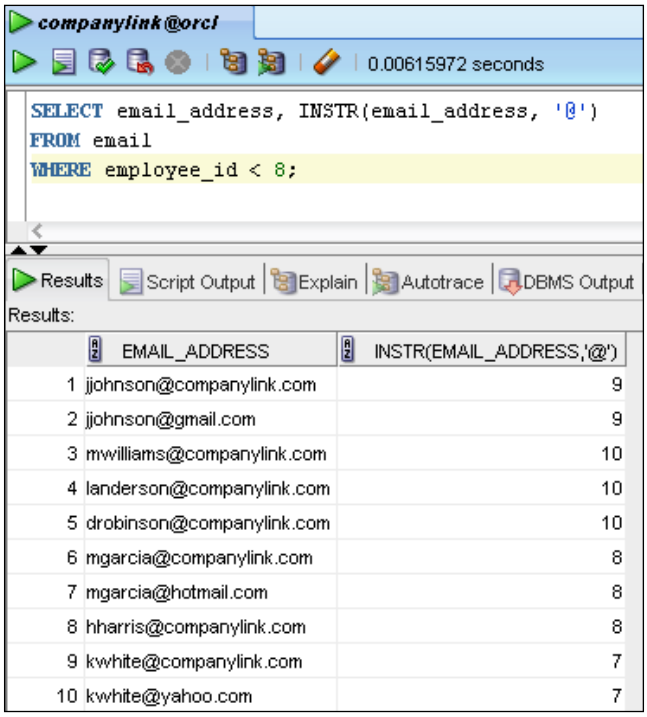
In short, we could say `SUBSTR()` does the following: take the `website_url` value, count over eight characters, then count over 19 more characters and return everything in between.

Although we normally provide positive integers for the starting character position in a `SUBSTR()` function, we can also use negative integers. The effect of this is to *count backward* in the string to get the starting point. Say, for instance, that we want to retrieve the last three characters from the domain name (known as the *top-level domain*) from each employee's e-mail address. To do this, we could designate a **-3** as the starting position in a `SUBSTR()` function, as shown in the following query. In this example, we count backwards three characters for the starting position and then move forward three characters to mark the end of the substring.



INSTR()

As we begin to make use of more advanced functions such as `SUBSTR()`, we may begin to see some of its limitations. For instance, what if we were asked to return the portion of an e-mail address *before* the '@' symbol (known as the *local-part*). This portion of an e-mail address is often the username of the user's e-mail account. Clearly, since we are retrieving a portion of the overall string value for an e-mail address, we will need to utilize `SUBSTR()` whose start position is 1. The difficulty, however, is that `SUBSTR()` requires us to count over by a known number of characters for the end position. Since each e-mail username can be a different length, we cannot accurately pinpoint what the end position should be. For situations such as this, we can use the `INSTR()` function to locate the position value of certain characters within a string, such as the '@' sign. The query in the following example can be used to locate the position of the '@' sign in each employee's e-mail address.



	EMAIL_ADDRESS	INSTR(EMAIL_ADDRESS,'@')
1	jjohnson@companylink.com	9
2	jjohnson@gmail.com	9
3	mwilliams@companylink.com	10
4	landerson@companylink.com	10
5	drobinson@companylink.com	10
6	mgarcia@companylink.com	8
7	mgarcia@hotmail.com	8
8	hharris@companylink.com	8
9	kwhite@companylink.com	7
10	kwhite@yahoo.com	7

The previous statement could be read as, *Locate the position of the first occurrence of the symbol '@' in the e-mail address*. Of course, it may be difficult to see the usefulness of this, since the value returned is only the position of the character in question. However, we will see in the next section that `INSTR()` can be combined with other functions in ways that make this extremely useful.

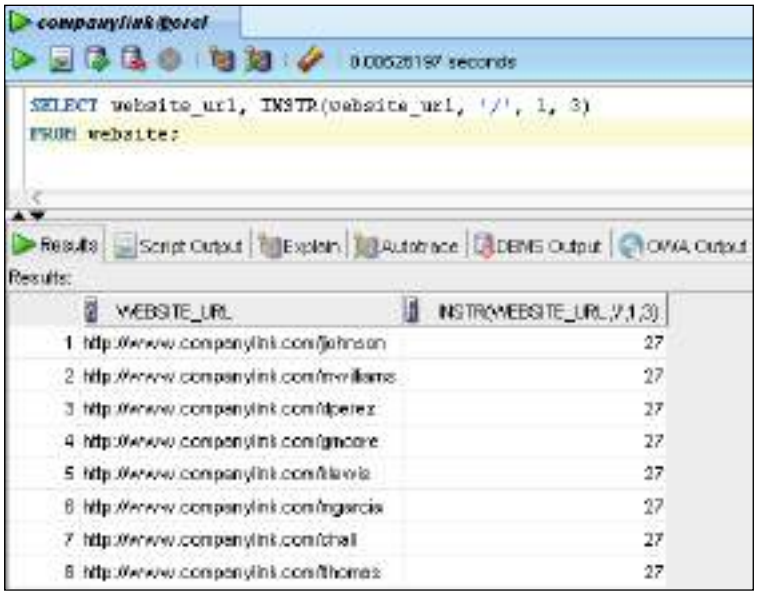
Although the previous `INSTR()` example takes only two arguments, it can also take two other optional arguments for even more control. An expanded syntax tree of the function itself with optional arguments and descriptions is shown as follows:

```
INSTR(column_expression, search_character, starting_position,
  occurrence_number)
```

where:

- `column_expression` = the column value to be searched
- `search_character` = the character for which to search
- `starting_position` = the position to begin the search; either a positive number to begin to the left, or a negative number to begin to the right
- `occurrence_number` = the number of occurrence for the searched character

Thus, not only can we specify which character for which to search, but also, optionally, we can designate where in the string we want to begin the search and even specify which occurrence of a particular character we wish to find. For example, say that we're tasked with writing a SQL statement that returns the portion of the string *after* the domain name in each website. We could say that we want the portion of the string following the '/' character, but it is actually the third occurrence of that character, since `http://` has two occurrences of the symbol as well. To find the position of the third '/' symbol, we could use the `INSTR()` function as shown in the following screenshot:



As we can see, in our example, the third occurrence of the '/' is found at the 27th character for each of our websites. However, as our Companylink site grows, it can accommodate URLs from other domains as well. Read simply, our `INSTR()` function searches the `website_url` column for the third occurrence of the '/', starting at the first character.

Exploring nested functions

We mentioned in our last section that in order for the `INSTR()` function to be useful, it would need to be combined with another function, such as `SUBSTR()`. We can do this with **nested functions**—functions that execute inside of other functions. Nested functions can be written many levels deep and be extremely complex, but we can interpret them, provided we follow some simple rules. First, remember to read the functions separately, rather than looking at the statement as a whole. Second, always interpret the innermost functions first. The following example shows the text of a statement using a `SUBSTR()` function with an embedded `INSTR()` function. The `INSTR()` is evaluated first and is used to *drive* one of the arguments for the `SUBSTR()`.

```
SELECT email_address,
SUBSTR(email_address, 1, (INSTR(email_address, '@')-1))
FROM email;
```

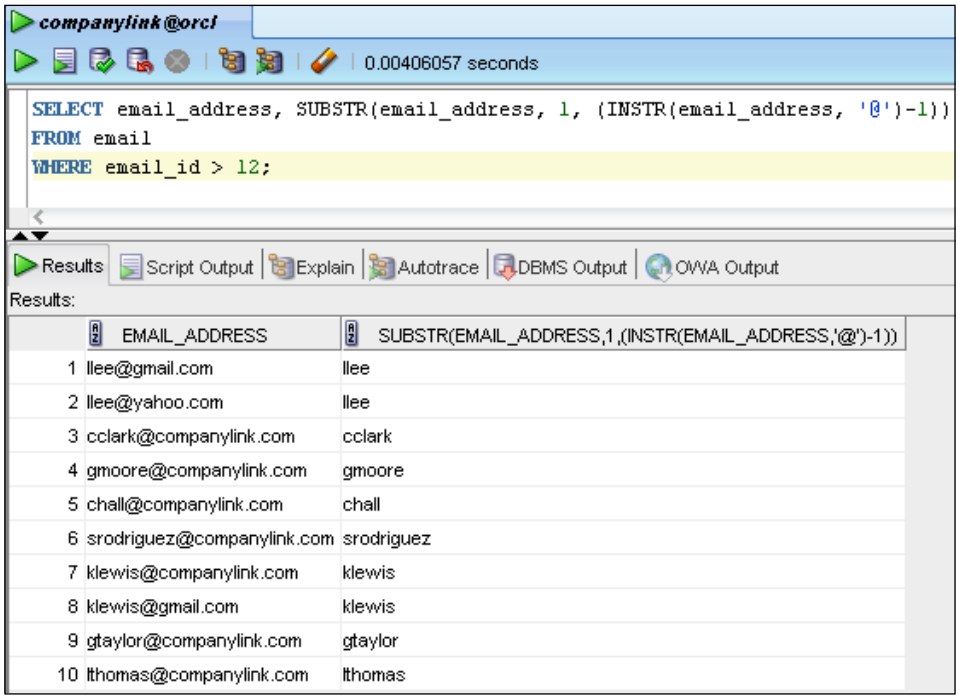
Rather than running the statement, let's just evaluate it for ourselves a step at a time. Following our rules, we evaluate the innermost functions first. The `INSTR()` function searches the `email_address` column for the first occurrence of the '@' sign. For the first row, this evaluates to the integer `9`, for the ninth place. After this evaluation, our statement would look like the one shown in the following syntax example:

```
SELECT email_address,
SUBSTR(email_address, 1, (9-1))
FROM email;
```

This is certainly easier to read. Next, we subtract one, since we want the ending position to be *one character before* the '@' symbol—not at the symbol itself. Without subtracting one, our e-mail account names would be displayed with the symbol included. Our statement now evaluates (for the first row) as shown in the following code.

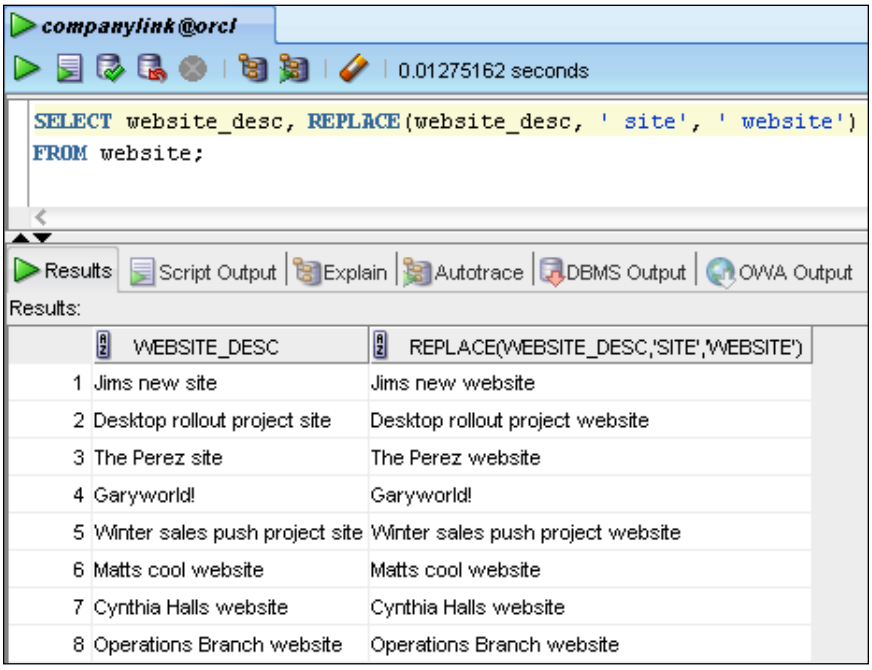
```
SELECT email_address,
SUBSTR(email_address, 1, 8)
FROM email;
```

Our statement now contains only a simple `SUBSTR()` function and can be evaluated accordingly. Remember that this particular example is only true for the first row, but subsequent rows are evaluated in the same way. The actual statement can do this recursively for each row of the table. The following screenshot shows the full statement and its execution:



Substituting values with REPLACE()

In SQL, it is often useful to be able to use the typical *search and replace* functionality present in many programs. The `REPLACE()` function provides the ability to search through a string value and replace any set of characters with another set. `REPLACE()` takes three arguments: the value to be searched, the string to replace, and the string with which to replace it. Say, for example, that we want to standardize the way that websites are named in our `Companylink` database. In the `website_desc` column, some websites are named `website` and others are named `site`; a situation we need to rectify by standardizing on the term `website`. Before we permanently alter the data, we want to see the data before and after the change. We can use the `REPLACE()` function to see the effects of this, as shown in the following example:



	WEBSITE_DESC	REPLACE(WEBSITE_DESC,'SITE','WEBSITE')
1	Jims new site	Jims new website
2	Desktop rollout project site	Desktop rollout project website
3	The Perez site	The Perez website
4	Garyworld!	Garyworld!
5	Winter sales push project site	Winter sales push project website
6	Matts cool website	Matts cool website
7	Cynthia Halls website	Cynthia Halls website
8	Operations Branch website	Operations Branch website

Our `REPLACE()` function takes in the `website_desc`, searches for `site,` and replaces it with `website`. Notice the leading space that is present in each of the strings, `site` and `website`. If we neglect this space, we will also substitute each occurrence of the string anywhere that `website` exists, transforming `website` into the string `webwebsite`. Remember that within single quotes, Oracle maintains capitalization. This is a good reminder that the `REPLACE()` function is strictly literal in its interpretation of our inputs.

Handling DATE functions

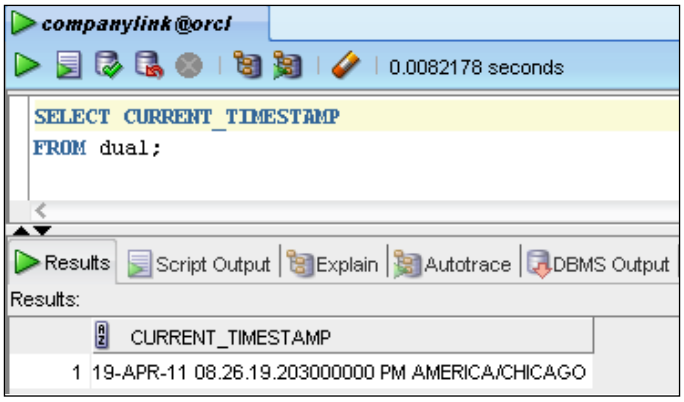
Handling values with the DATE datatype in SQL presents several challenges. In Oracle and many other database systems, this is primarily because a date is neither a string value nor a numeric value. Yet, using the proper functions, we can transform values of the DATE datatype into string values and even do date arithmetic. To see how this works, let's begin by returning to the subject of `SYSDATE` that we mentioned in *Chapter 2*, *SQL SELECT Statments*.

Distinguishing SYSDATE and CURRENT_TIMESTAMP

As we mentioned in *Chapter 2*, *SQL SELECT Statments*, `SYSDATE` is an Oracle pseudo-column—it does not represent an actual column value, but rather returns a system value generated by Oracle. In the case of `SYSDATE`, it returns the current date and, if we instruct it to do so, the time as well. Let's look at an example using `SYSDATE`, as shown in the following example:

If you run the command listed previously, your results will differ. That's because `SYSDATE` returns the current date when the statement is run. Since you are running the statement after this book has been published (unless you're exceptionally good at time travel) your system date will be different. If you run the statement tomorrow, it will be different still. It is important to understand that `SYSDATE` *returns the current date from the perspective of the database to which we are connected*. If you are running this statement from SQL Developer against an installation of Oracle that you have, yourself, done (as suggested in *Chapter 1, SQL and Relational Databases*), both will be on the same server. The database inherits the same time as that of the server it resides upon. Your installation will therefore constitute both the client (SQL Developer) and the server (Oracle). If we were to connect to a database in a different time zone, say, one that is across the International Date Line, our `SYSDATE` would actually differ from our local date.

In the event that we wish to return the date from a different perspective, we could use the `CURRENT_TIMESTAMP` pseudo-column, as shown in the next example:



Notice the difference in the value returned. Again, it will be different from the value displayed in this book. There is also much more information returned. In addition to the date, we see time information down to fractions of seconds and an offset to calculate time zone. This is because `CURRENT_TIMESTAMP` displays its information using a different datatype—the TIMESTAMP datatype. `CURRENT_TIMESTAMP` retrieves its information differently than `SYSDATE`. `CURRENT_TIMESTAMP` *returns the current date and time from the perspective of the user's session on the client machine that is connecting to the database.*

SQL in the real world

Although DATE is still the most widely-used datatype to store date and time information, the TIMESTAMP datatype is growing in popularity due to its ability to store time information to a greater numeric precision. Oracle also has other similar datatypes that are designed to be used in environments where time zone information is important.

Another difference between these two pseudo-columns is the manner in which they are stored. While a discussion of the storage methods of the `TIMESTAMP` datatype are beyond the scope of this book, it is useful to understand how values with the DATE datatype are stored. In Oracle, the DATE datatype stores seven bytes that contain the amount of time that begins at a point long in the past—January 1, 4712 BC, to be exact. That date can be considered *date zero*. Oracle uses internal mechanisms to interpret the information stored in any DATE value against *date zero* to calculate date and time and display it accordingly. Because DATE values are stored this way, we can use date functions to display dates in many different ways.

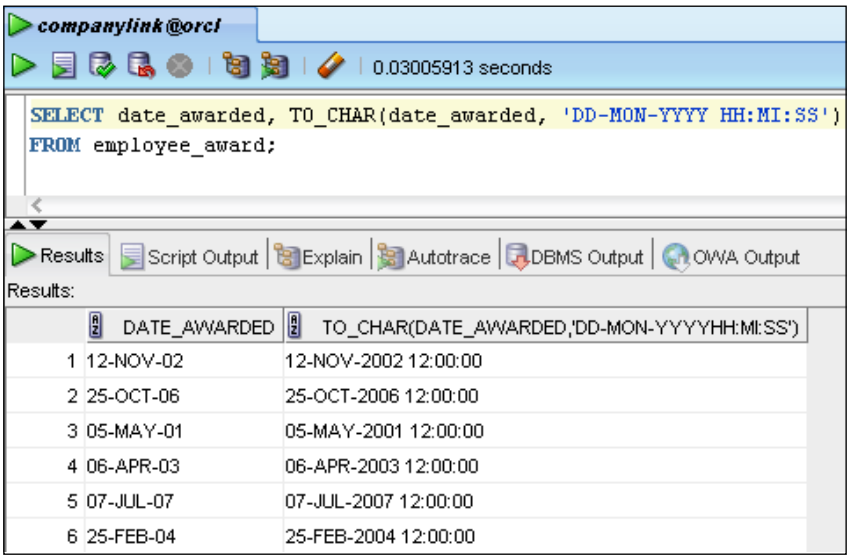
Oracle has a default format for displaying this date information. You can see it in the previous example. The default format is DD-MON-YY, where DD is the two-digit day, MON is the three-digit month abbreviation, and YY is the two-digit year. However, there are obviously many situations in which we would want to display dates in different formats. For this, we can use datatype conversion functions.

Utilizing datatype conversion functions

Datatype conversion functions are so named because they convert values of one datatype to another. This is relevant to our discussion of dates since displaying a date in a format different than the default format requires converting it to a character string value. As we will see, we can use datatype conversion functions to convert date values to strings, string values to dates, and even numeric values to strings.

Using date to character conversion with TO_CHAR

One of the most common uses of datatype conversion functions is to convert a date into a character string to change the format in which it is displayed. This type of conversion requires both a value to convert and instructions for how to convert it. The instructions for converting a date value are passed to a function called `TO_CHAR()`, or *to character*, in the form of a **format mask**, which details how a date value should be displayed. An example is shown in following query:



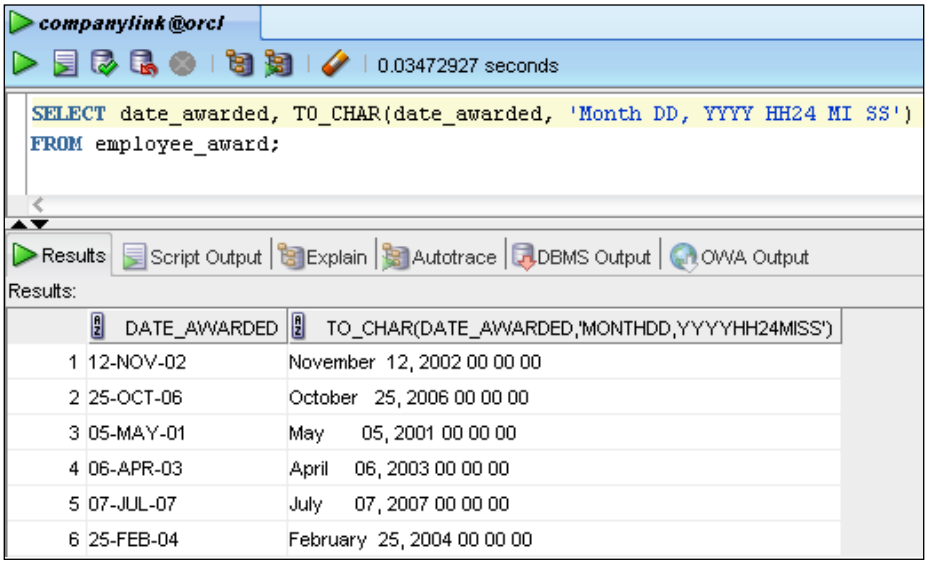
Even as complex as this statement looks, notice that as with other functions, we are simply passing arguments to the `TO_CHAR()` function. The first argument is the value of the column in question, `date_awarded`. We also display this column prior to the converted value for reference. The second argument is our format mask (also known as a format model), which provides instructions as to how the value should be displayed. `TO_CHAR()` converts the DATE value to a character string value and displays it as instructed by the format mask. The format mask by itself is shown as follows:

```
'DD-MON-YYYY HH:MI:SS'
```

Although this may look confusing, we simply need to break it down into its individual parts in order to interpret it. We first notice that the format mask is enclosed in single quotes—this is always the case. Next, we note that the individual elements are separated by either dashes (-) or colons (:). These characters are retained in the date format that is displayed. If we wish (as we will see in later examples), other characters can be substituted for the ones in our example. However, the characters we use in the format mask will be the ones displayed in our output. For example, the time information is delimited by colons, which is the way time is commonly displayed.

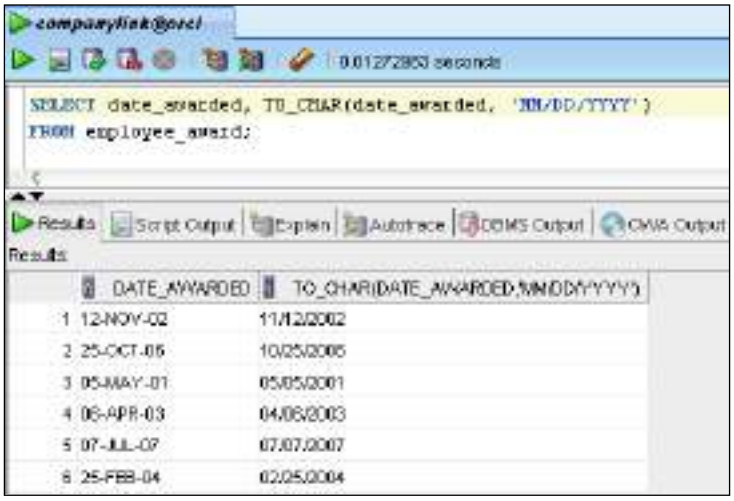
Lastly, we examine the individual elements of the format mask themselves. Each of these elements reference a particular part of the date/time information. `DD` instructs the format mask to display a two-digit day, between 01 and 31. `MON` refers to a three-digit month abbreviation, such as JAN or FEB. `YYYY` represents a four-digit year. Next is the time information. `HH` instructs the mask to display the number of hours stored in the value in two digits, such as 12 or 03. Note that this will display the hours with respect to a 12-hour clock, using AM or PM as needed. `MI` refers to the number of minutes in two digits. Finally, `SS` represents the number of seconds in our DATE.

Using the `TO_CHAR()` function, we can display date and time information in a variety of ways. In the following example, we rewrite the previous statement to display the same dates in an entirely different format by changing only the format mask:



Here, we have used a different element, Month, to display the full name of the month in question. Notice that the month name is displayed in mixed case. Our format mask element is mixed case, so our month is shown as such. Remember that in Oracle, information that is stored inside single quotes is case-sensitive. This applies to our format mask elements, as well. Also note that we have changed the order in which date information is displayed, with month information coming before day information. We have also used a comma to delimit our date information and spaces to delimit our time information. As a result of the different format mask, our date data is displayed in a different manner.

When using format masks with `TO_CHAR()`, it is not necessary to include both date and time. Our format mask can specify one or the other, as shown in the following screenshot:



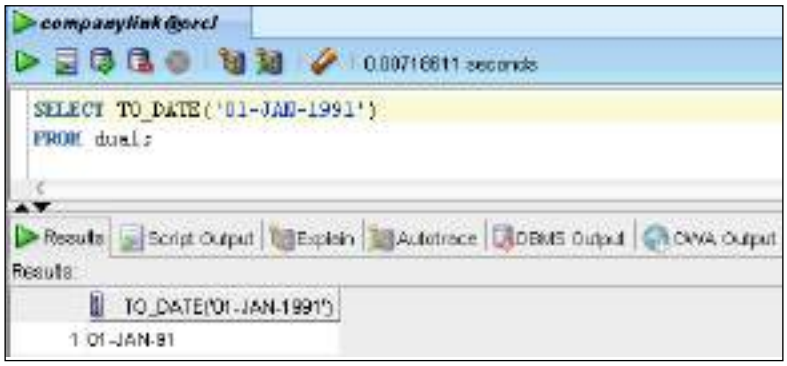
Here, the `date_awarded` column is shown with only days, months, and years. The first, unmodified column, also shows this information, but does so in the default format we mentioned earlier.

There are a vast number of elements that can be used in a format mask. The following table shows several of the more common ones:

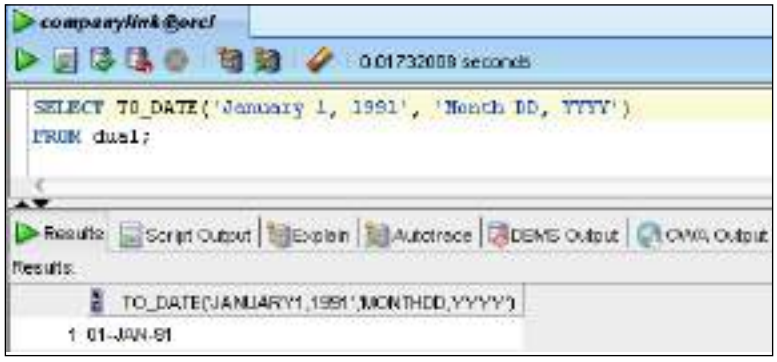
Element	Example	Description
DD	22	Two digit day of the month
Day	Tuesday	Mixed case day of the week
DAY	TUESDAY	Capitalized day of the week
D	3 (Tuesday)	Numeric value for day of the week
DDD	081	Numeric value for day of the year
DY	TUE	Three digit abbreviation for day of the week
DL	Tuesday, March 22, 2011	Date in long format
MM	03	Two digit month of the year
Month	March	Mixed case month of the year
MONTH	MARCH	Capitalized month of the year
MON	MAR	Three digit abbreviation for month of the year
YY	11	Two digit year
YYYY	2011	Four digit year
Year	Twenty Eleven	Mixed case name of the year
YEAR	TWENTY ELEVEN	Capitalized name of the year
HH	10	Number of hours on the twelve hour clock
HH24	22	Number of hours on the twenty four hour clock
MI	14	Number of minutes
SS	47	Number of seconds
SSSS	0505	Number of seconds past midnight
AM/PM	PM	Meridian indicator on the twelve hour clock
CC	21	Numeric value for century
BC/AD	AD	Epoch indicator

Converting characters to dates with TO_DATE()

Just as we can convert a date to a character string with `TO_CHAR()`, we can likewise take a character string and convert it into a date. The `TO_DATE()` function takes a date-oriented string and converts it into a value with a datatype of DATE. Obviously, the character string that we pass to `TO_DATE()` must be a date of some type—we cannot simply pass a random string of characters and expect `TO_DATE()` to convert it properly. An example is shown as follows:



In this example, we pass the string '01-JAN-1991' to the `TO_DATE()` function. It is then converted and displayed in the default format for the database. In this statement, it is crucial that the string passed to the function must be in a format that Oracle can automatically recognize as a date. However, this can be fairly limiting. It is possible that we may want to enter string dates in a format that differs from the database's default format. For this, we turn again to format masks. Any string that is passed to `TO_DATE()` can accompany a format mask that defines how the string should be converted. Fortunately, the type of format mask for this operation is the same as the ones used with `TO_CHAR()`. In the next example, we pass the character string **'January 1, 1991'** to the `TO_DATE()` function and instruct the function to convert it using a format mask that specifies how the string value is formatted.



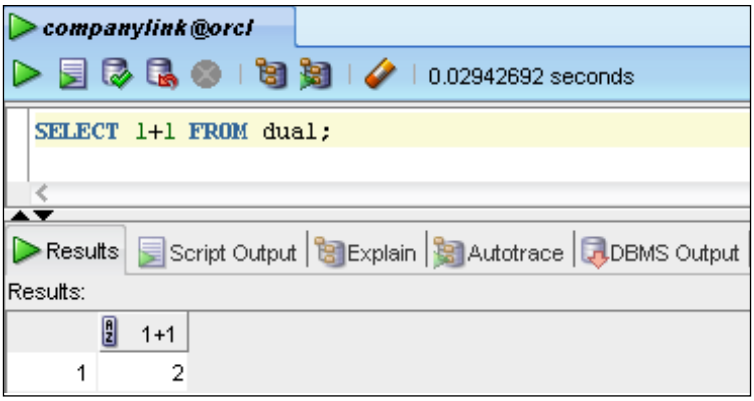
In this example, we pass the value **'January 1, 1991'**, a value that Oracle cannot implicitly recognize as a date, to the `TO_DATE()` function. Because it is not immediately recognizable, we must include a format mask as the second argument. This mask will instruct the function as to how to *read* the date—month value, space, day value, comma, space, year value. It then returns a date in the default database format.

SQL in the real world

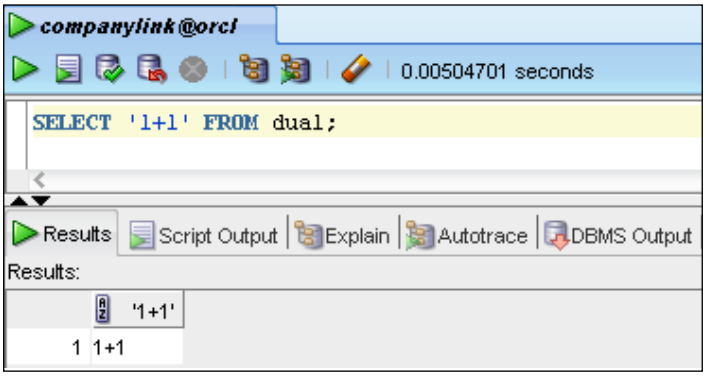
The SQL function `TO_DATE()` can be extremely useful in real-world situations that require dates to be converted from older database systems. In many of these systems, dates are stored as character string values instead of anything resembling Oracle's DATE datatype. The `TO_DATE()` function was used extensively in the late 1990s in solutions to the "Year 2000 problem" to convert older style character dates to Y2K-compliant dates.

Converting numbers using TO_NUMBER()

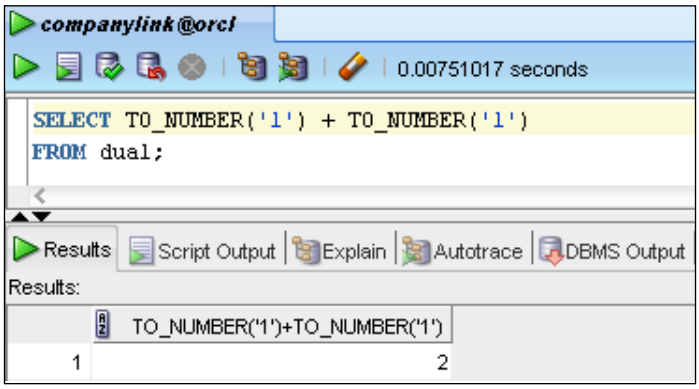
Our last datatype conversion function does not deal with date values. Rather, the `TO_NUMBER()` function is used to convert character values to numeric ones, often for the purpose of taking character values and performing an arithmetic operation. Let's look at a simple example. We know from *Chapter 2, SQL SELECT Statments,* that we can use arithmetic expressions with the DUAL table in order to do simple math, as we see in the following example:



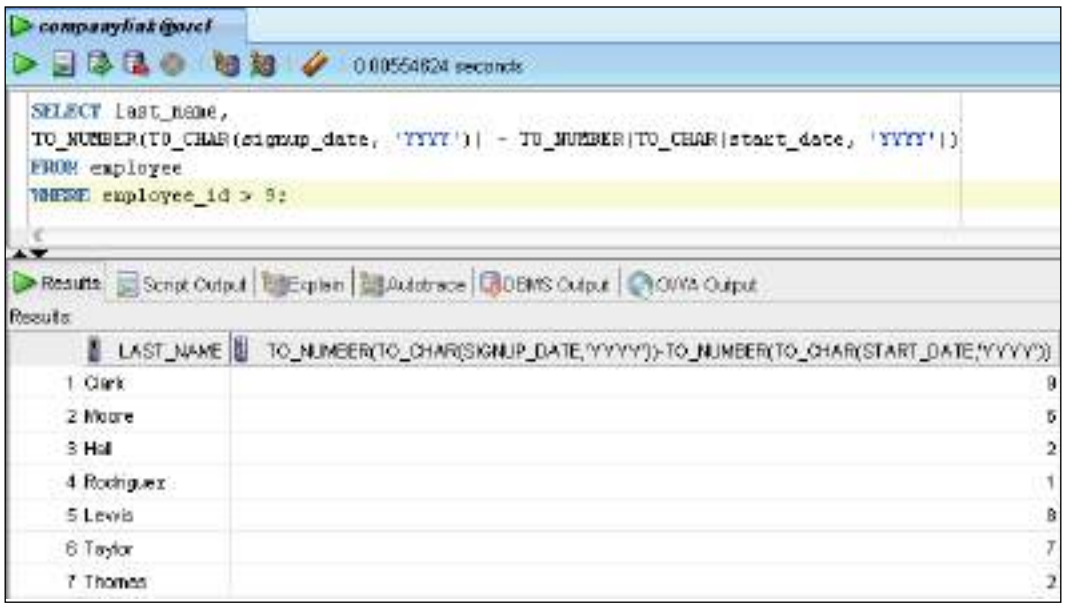
The value returned, of course, is 2. This works because each of the values is numeric. However, if the values were enclosed in single quotes, that is to say they represent character values, the result is quite different, as we can see in the following screenshot:



The result is not the arithmetic outcome we might have expected. This is because Oracle evaluates anything within single quotes as a character string. If our values were strings that we wished to add, we would first need to convert them using `TO_NUMBER()`, as shown in the next screenshot. Each of the character strings **'1'** is first converted to a numeric value and then added together.



While this example in itself may not seem particularly useful, it does demonstrate that string values can be used in arithmetic expressions. However, used in conjunction with our `TO_CHAR()` function, `TO_NUMBER()` can be put to good use. Say, for example, that we needed to display a rough approximation of the number of years between the date that an employee was hired and the date that they signed up for Companylink. We could use `TO_NUMBER()` in the example shown in the next screenshot to do this. In real world terms, we might interpret this query as, *Find the number of years between hire and sign up*.



Here, we first extract only the year of each employee's signup and hire date using `TO_CHAR()` with a format mask of `'YYYY'`. This evaluates the two values for each, which are subtracted. However, since we have used `TO_CHAR()` to extract the year values, they are character values that require a numeric conversion. Thus, we place each value within a `TO_NUMBER()` function in order to convert them. Also note that these are approximate values, since the month and day values are stripped from the dates and are not considered.

SQL in the real world

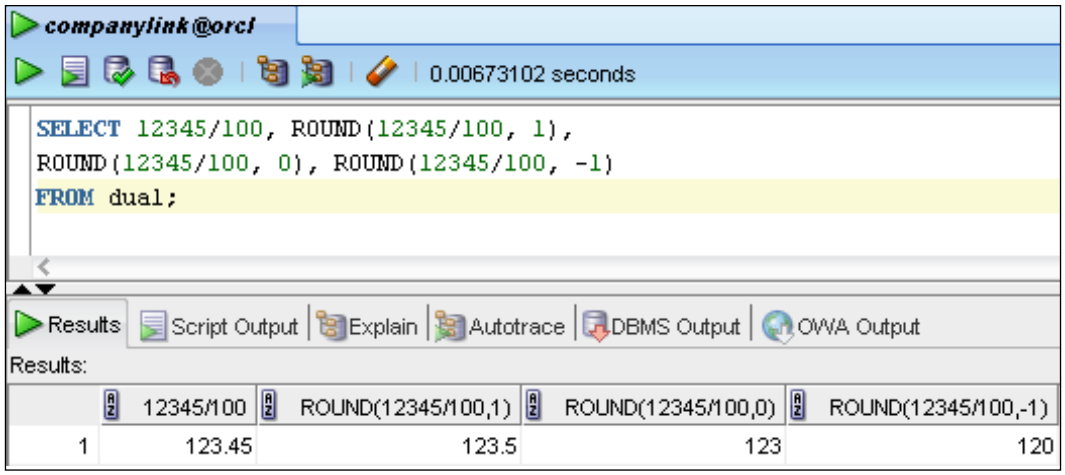
Oracle can do a certain amount of automatic datatype conversion, but it is often considered bad syntax to rely on it. For instance, the previous example can be successfully executed without the `TO_NUMBER()` functions, but many coding standards would reject the statement as unclear.

Using arithmetic functions

Just as with the string and date functions, we have an entire set of functions that deals strictly with numeric values and arithmetic operations. We will look at a few of the more common examples here, but there are many different functions you can call in order to manipulate numbers in Oracle.

ROUND()

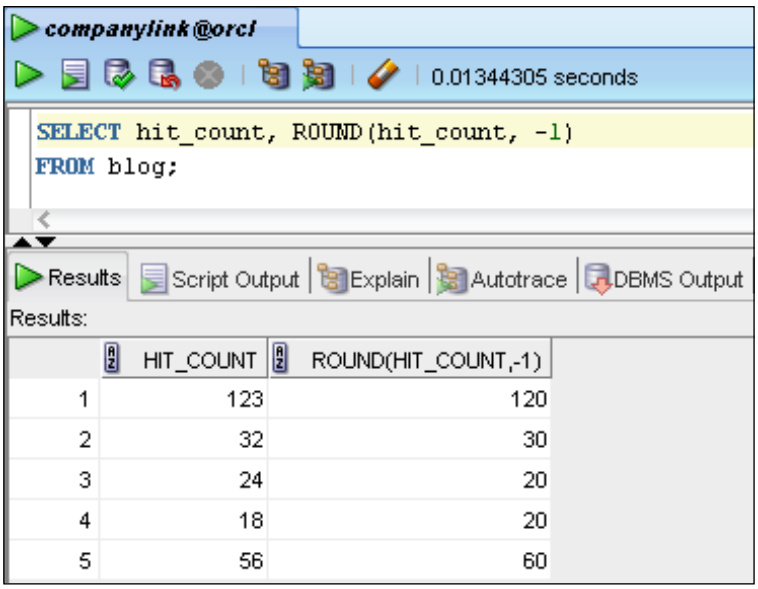
The `ROUND()` function is straightforward in its purpose—it rounds numeric values according to standard mathematic rules. Digits less than five are rounded down, and values greater than or equal to five are rounded up. However, because of the complexity involved in specifying the decimal place to which values should be rounded, learning to manipulate `ROUND()` can be tricky. Let's first look at an example using numeric literals, as shown in the following screenshot:



In this example, we are using the quotient that results from dividing **12345** and **100**, the numeric value **123.45**. The `ROUND()` function takes two inputs: the value to be rounded and the decimal place to which to round it, denoted by an integer value. If the decimal place is 1, our value will be rounded to the nearest tenth; if it is 2, the value is rounded to the nearest hundredth, and so on. We can, however, also use a zero or negative value for the decimal place. If the decimal place is 0, the value is rounded to the nearest ones place. If -1, it is rounded to the nearest tens. The following chart shows the effect of using several different decimal places in our `ROUND()` function. The following chart should help distinguish the effects of using various values for the decimal place in the `ROUND()` function.

x	ROUND(x,1)	ROUND(x,2)	ROUND(x, 0)	ROUND(x,-1)	ROUND(x,-2)
105.36	105.4	105.36	105	110	100
13.88	13.9	13.88	14	10	0
7.34	7.3	7.34	7	10	0
0.52	0.5	0.52	1	0	0

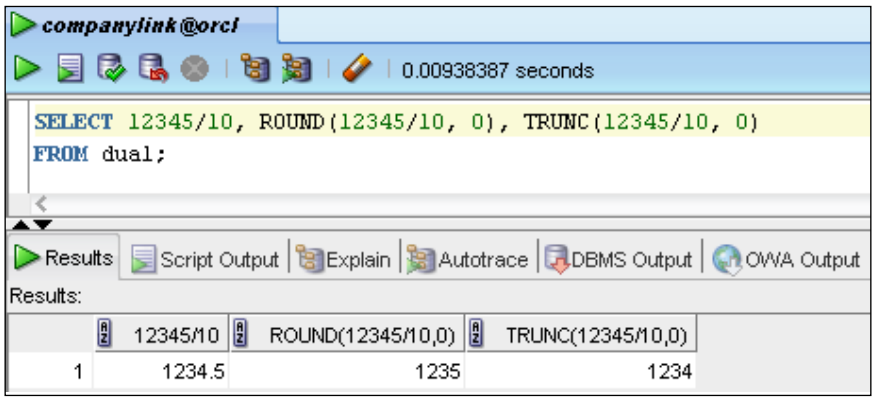
Say, for example, that we need to do some calculations on the number of hits to employee's blogs. We could use the `ROUND()` function to do this, as shown in the following example. By passing the value -1 to the `ROUND()` function, we round each blog's hit count to the nearest tens place.



	HIT_COUNT	ROUND(HIT_COUNT,-1)
1	123	120
2	32	30
3	24	20
4	18	20
5	56	60

TRUNC()

The `TRUNC()` function performs an operation similar to that of `ROUND()`. However, instead of rounding the value in question to the nearest specified decimal place, `TRUNC()` merely truncates the value without rounding. A comparison of the `ROUND()` and `TRUNC()` functions is shown in the following screenshot:

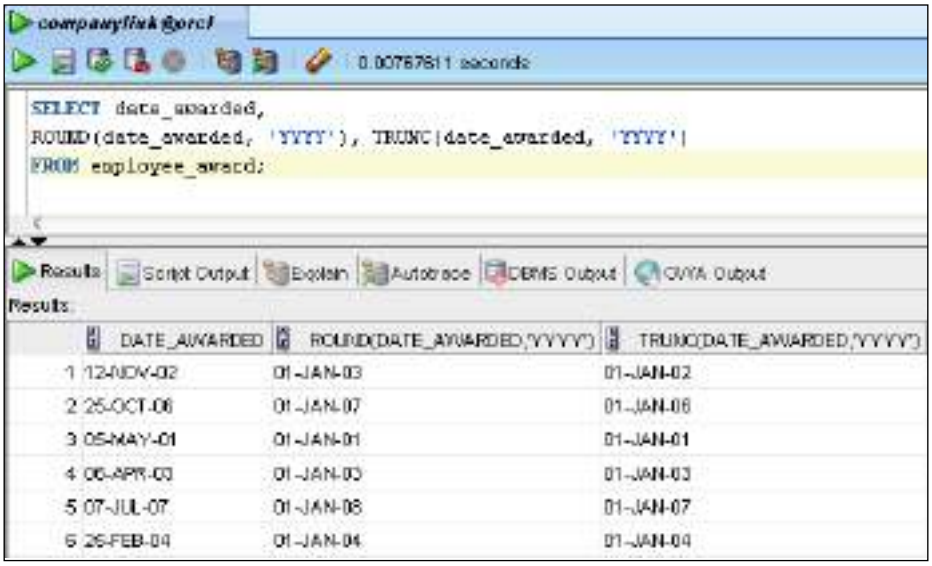


As you can see, our base number is **12345/10**, or **1234.5**. While `ROUND()` behaves as we have seen, rounding the number to 1235, the `TRUNC()` function simply truncates the decimal place and returns a value of 1234.

Using ROUND() and TRUNC() with dates

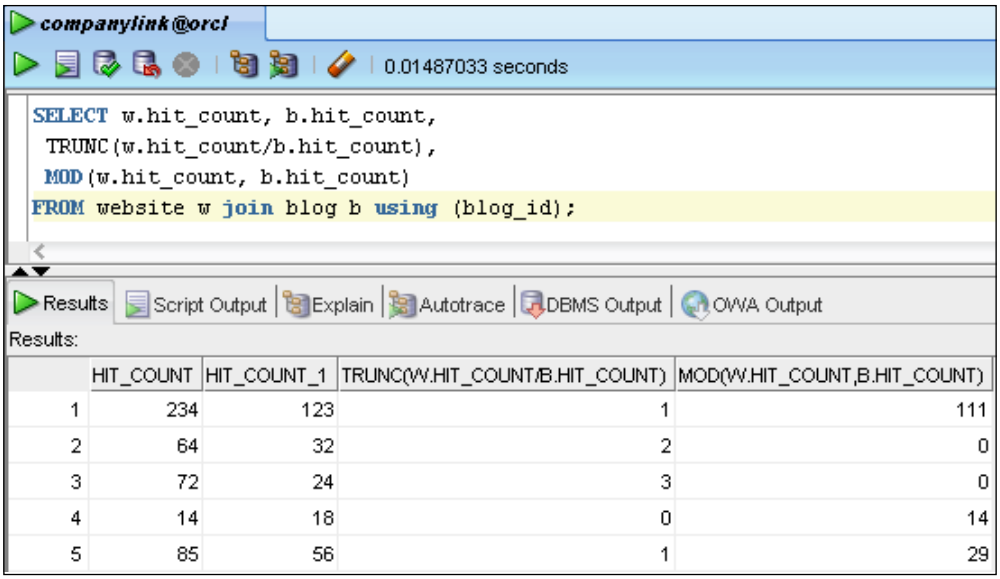
Both `ROUND()` and `TRUNC()` can be used to modify date values as well as numbers. When used with dates, these functions take a date value and a format mask as arguments. It is the format mask that instructs the function as to the date element that should be rounded. For instance, if we round a date with a format mask for the year of **'YYYY'**, it is rounded to January 1 of the nearest year. So, if the date occurs during the second half of the year, it is rounded up to January 1 of the following year. If the date occurs in the first half of the year, it is rounded down to January 1 of that same year. Rounding with months works similarly.

The `TRUNC()` function takes a date and returns the previous corresponding date, depending on what is used as the format mask. An example is shown in the following screenshot:



MOD()

The `MOD()` function works the same as the modulus operator in mathematics. It takes two values, a dividend and a divisor, and returns the remainder after dividing the two. An example is shown in the following query. The statement takes the hit counts for the website and blog tables from the `Companylink` database, divides them, and returns the remainder, or modulus. This statement returns, in order, the dividend, the divisor, the whole number quotient, and the remainder. We use `TRUNC()` in order to remove the fractional part of the quotient.



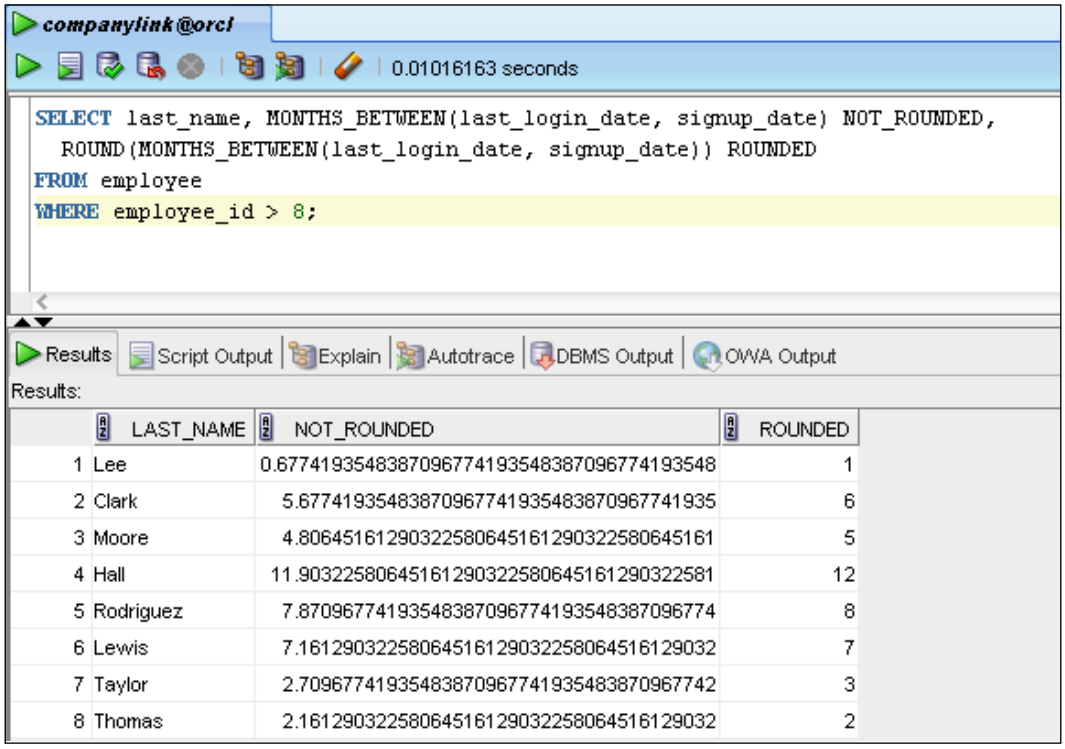
	HIT_COUNT	HIT_COUNT_1	TRUNC(W.HIT_COUNT/B.HIT_COUNT)	MOD(W.HIT_COUNT,B.HIT_COUNT)
1	234	123	1	111
2	64	32	2	0
3	72	24	3	0
4	14	18	0	14
5	85	56	1	29

Understanding date arithmetic functions

Oracle includes many built-in functions that can perform **date arithmetic**—the process of adding and subtracting dates. Date arithmetic is often a challenge in programming languages because it attempts to apply *base ten* arithmetic operations to dates, which do not easily conform to such a numeric system. In Oracle's implementation of SQL, however, this is easily accomplished using date arithmetic functions.

MONTHS_BETWEEN()

The `MONTHS_BETWEEN()` function returns a numeric value that represents the number of months between two dates. Its syntax and use are shown in the following example:

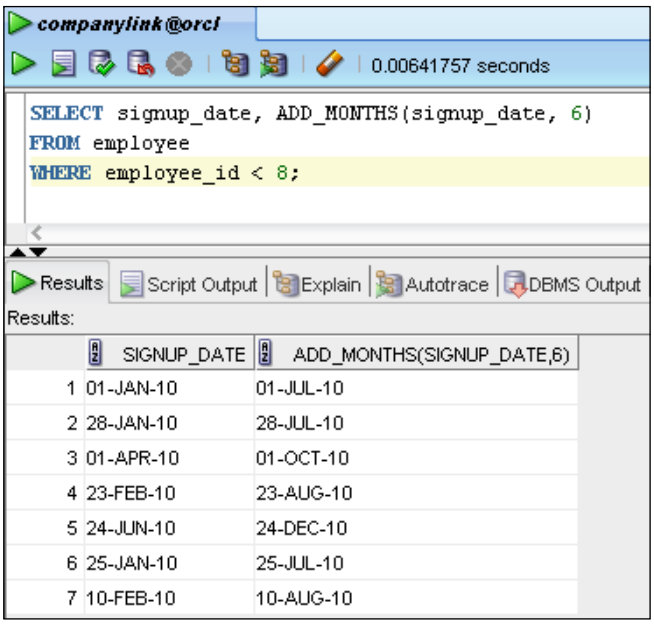


	LAST_NAME	NOT_ROUNDED	ROUNDED
1	Lee	0.677419354838709677419354838709677419354	1
2	Clark	5.677419354838709677419354838709677419354	6
3	Moore	4.806451612903225806451612903225806451612	5
4	Hall	11.903225806451612903225806451612903225	12
5	Rodriguez	7.870967741935483870967741935483870967742	8
6	Lewis	7.161290322580645161290322580645161290322	7
7	Taylor	2.709677419354838709677419354838709677419	3
8	Thomas	2.161290322580645161290322580645161290322	2

We notice that the value returned is a true numeric value that represents the number of fractional months between the two dates that are passed as arguments. In order to make this output more readable, we've shown two columns that utilize the `MONTHS_BETWEEN()` function. The first shows the non-rounded, true numeric value, and the second applies the `ROUND()` function as an outer function that rounds the decimal value into a more readable integer value. We also place aliases on both columns to simplify the column headings.

ADD_MONTHS()

The `ADD_MONTHS()` function is used to simply add a specified number of months to a given date. The function takes two arguments: the date itself and the number of months to add. In the following screenshot, we add six months to each employee's `signup_date`.

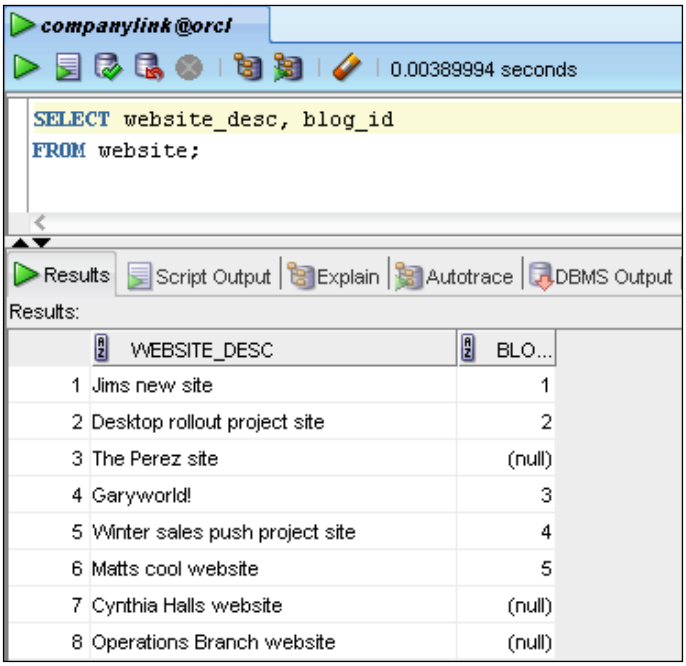


Examining functions that execute conditional retrieval

Within Oracle, we have another set of functions at our disposal that doesn't easily fit into the categories that we've seen thus far. These functions all address, to some degree, the need for *if...then* logic within SQL. The first two of these functions provide special ways in dealing with NULL values.

NVL()

The NULL value is an essential part of SQL. Without it, we would have no way to express the lack of a value within a row. Still, NULL values can present problems, particularly when attempting to use them with arithmetic expressions. For instance, recall an example from *Chapter 2, SQL SELECT Statments,* where we introduced the topic of nulls. In it, we attempted to divide the number 100 by a NULL. The result was another NULL value. Most often, NULL values create problems when they are not expected. For instance, say we want to create a small report that lists the website description for each employee website and its associated `blog_id`. We could do a simple query such as the one shown in the following example:

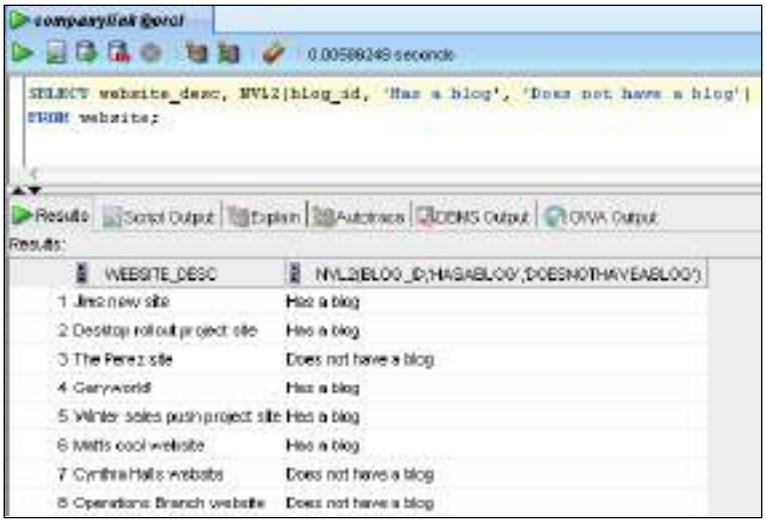


	WEBSITE_DESC	BLO...
1	Jims new site	1
2	Desktop rollout project site	2
3	The Perez site	(null)
4	Garyworld!	3
5	Winter sales push project site	4
6	Matts cool website	5
7	Cynthia Halls website	(null)
8	Operations Branch website	(null)

If we so desired, we could replace these NULL values with another number, say a zero, in order to tidy up the report. To do so, we use the `NVL()` function, which takes two arguments. The first is the value to evaluate. The second is a substitution value in the event that the first value is NULL. Thus, we could read the example in the following screenshot as, *Display website description and blog ID. If the blog ID is NULL, display a zero.*

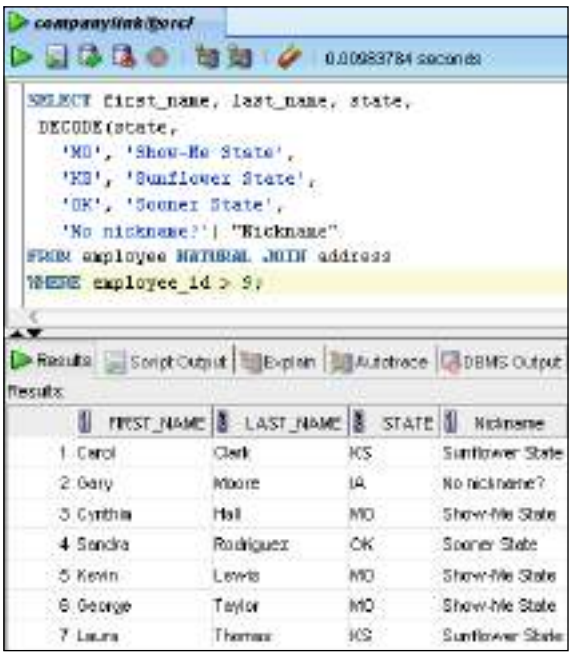
NVL2()

The `NVL2()` function is an extension of `NVL()` that allows the programmer more flexibility. `NVL2()` takes three arguments. The first is the value to be evaluated. The second is the value to display if the first value is *not* a null. The third is the value to display if the first value *is* a null. Using `NVL2()`, we can conditionally evaluate a list of values and display one value if the original value is not null and a different one if it is null. To return to our earlier example, let's say that we still want to display a list of employee websites. But, now we are not concerned with showing the blog ID. However, we do still want to show whether or not each website *has* a blog associated with it. To show this, we could use the `NVL2()` function as shown in the next example:



DECODE()

Of the three conditional retrieval functions we examine here, `DECODE()` is the most similar to the true *if...then* statements found in third-generation programming languages. The `DECODE()` function is different in that it evaluates most of its arguments in pairs. The first argument passed to the function is the value to be evaluated. Thereafter, we list pairs of values. The first of the pair is the value to match. The second of the pair is the value to display if a match is found. The final value passed to the function is the value to display if no match is found for the evaluated value. Say, for instance, that we want to display the name of each employee and the nickname of the state they are from. Although we don't store state nickname information in our `Companylink` database, we do store the abbreviation of each employee's state in their address. Thus, to complete the task, we'll need to join the employee and address tables, then use a `DECODE()` function, as we see in the following example:



As we can see, those employees whose address is **MO** display a nickname of **Show-Me State**. Employees from other states in the list of values to match also show their state nicknames. **Gary Moore**, however, has an **IA** address, which is not found in our list of matching values. His entry, therefore, defaults to the last value in the list, which is 'No nickname?'.

SQL in the real world

Although many single-row functions can, with practice, become fairly intuitive and easy to remember, their sheer number can be overwhelming. SQL includes hundreds of built-in functions for daily use. Just remember that functions exist to make your coding easier. You may work your entire career and only use a fraction of them. Your best reference for the syntax of the many functions available to you is the SQL Language Quick Reference online book that is hosted at `http://technet.oracle.com`.

Summary

In this chapter, we've examined some of the many single-row functions available to you as a SQL coder. We've looked at the structure and purpose of functions, as well as the different types that exist. We have used string functions in data transformation, worked with numeric functions to accomplish arithmetic operations, and used datatype conversion functions to convert values between datatypes. We closed with a look at functions that conditionally retrieve data in a way similar to the *if...then* statements found in other third-generation programming languages.

Certification objectives covered

- Describe various types of functions available in SQL
- Use character, number, and date functions in `SELECT` statements
- Using Conversion Functions and Conditional Expressions
- Describe various types of conversion functions that are available in SQL
- Use the `TO_CHAR()`, `TO_NUMBER()`, and `TO_DATE()` conversion functions

You may think that there couldn't possibly be any more functions to see, but there are. In the next chapter, we'll look at the complement to single-row functions, multi-row functions, that let us process an entire rowset and return a single value. We'll also look at the last of the major clauses found in `SELECT` statements—`GROUP BY` and `HAVING`.

Test your knowledge

1. Functions that take a row of data as input and return a corresponding single value are known as
   a. multi-row functions
   b. single-row functions
   c. dual-row functions
   d. transformation functions

2 Given that the last name of the employee with an `employee_id` of `6` is `Harris`, what value is returned from the following query?

```
SELECT UPPER(last_name) FROM employee WHERE employee_id = 6;
```

   a. harris
   b. Harris
   c. HARRIS
   d. hARRIS

3. Given an employee named `Ken White` with an `employee_id` of `7`, what two values are returned from the following query?

```
SELECT INITCAP(first_name), LOWER(last_name)
FROM employee WHERE employee_id = 7;
```

   a. Ken White
   b. ken white
   c. kEN WHITE
   d. Ken white

4. What value is returned from the following query?

```
SELECT LENGTH((CONCAT('Learning SQL','is fun'))) FROM dual;
```

   a. Learning SQLis fun
   b. Learning SQL is fun
   c. 18
   d. 19

5. Given the following SQL statement, which statement is FALSE?

```
SELECT RPAD(project_name, 31, '*') FROM project;
```

   a. The padding characters appear on the right side of the `project_name` value.
   b. Any value for `project_name` that is 31 or more values will have no padding.
   c. The character that will be displayed as padding is the asterisk, `*`.
   d. Any value for `project_name` that is 20 characters in length will have 10 asterisks padded on the right side.

6. The function used to *trim* values from the left side of a value is
   a. RTRIM()
   b. RIGHTTRIM()
   c. LTRIM()
   d. INITCAP()

7. Given a value for `website_url` that is `http://www.companylink.com/dperez` where `website_id` is 3, what would be the output of the following statement?

```
SELECT SUBSTR(website_url, 6, 6)
FROM website
WHERE website_id = 3;
```

   a. www
   b. www.
   c. //www.
   d. //www.c

8. Given a value for `website_url` that is `http://www.companylink.com/dperez` where `website_id` is `3`, what would be the output of the following statement?

```
SELECT SUBSTR(website_url, -4, 8)
FROM website
WHERE website_id = 3;
```

   a. dperez
   b. perez
   c. erez
   d. erezhttp

9. Given a value for `website_url` that is `http://www.companylink.com/dperez` where `website_id` is `3`, what would be the output of the following statement?

```
SELECT INSTR(website_url, '.', 6, 2)
FROM website;
WHERE website_id = 3;
```

   a. 0
   b. 13
   c. 17
   d. 23

10. Given the following statement, what would be the output?

```
SELECT REPLACE(REPLACE('Follow the yellow brick road','o', '0'),
  'e', '3')
FROM dual;
```

   a. Follow the yellow brick road
   b. F0ll0w the yell0w brick r0ad
   c. F0ll0w th3 y3ll0w brick r0ad
   d. An error is generated. You cannot nest one `REPLACE()` function within another.

11. Given a `TO_CHAR` function that uses a format mask of

```
'DD-MONTH-YYYY HH24:MI:SS',
```

    which of the following dates could most closely resemble the possible output?

   a. JUNE 20, 2011, 13:02:01
   b. 18-March-2011, 02:22:09 PM
   c. 01-JAN-2011, 09:15:42
   d. 22-APRIL-2011, 15:22:37

12. Which format mask below would produce a date in the following format?

    October 27, 1999 @14.23.15

   a. 'MONTH DD, YYYY @HH:MI:SS'
   b. 'Month DD, YYYY @HH24:MM:SS'
   c. 'Month DD, YYYY @HH:MI:SS'
   d. 'Month DD, YYYY @HH24:MI:SS'

13. Which of the values below would be produced by the following statement?

```
SELECT TO_DATE('January 1, 1991', 'Month DD, YYYY')
FROM dual;
```

    a. January 1, 1991
    b. 01-JAN-91
    c. JAN 01, 1991
    d. 01/01/91

14. What would be the three values outputted, in order, from this statement?

```
SELECT ROUND(3.14159, 4),
TRUNC(3.14159, 3),
ROUND(TRUNC(3.14159, 4), 2)
FROM DUAL;
```

    a. 3.141, 3.14, 3.1
    b. 3.1415, 3.142, 3.14
    c. 3.1416, 3.142, 3.14
    d. 3.1416, 3.141, 3.14

15. Examine the data in the `employee` table. How many rows with a value of -1 would be produced if the following statement was executed?

```
SELECT NVL(project_id, -1) FROM employee;
```

    a. 0
    b. 1
    c. 3
    d. 4

16. Examine the data in the `employee` table. How many rows with a value of `Unassigned` would be produced if the following statement was executed?

```
SELECT project_id, NVL2(project_id, 'Project Assigned',
  'Unassigned')
FROM employee;
```

    a. 0
    b. 2
    c. 3
    d. 4

7
Aggregate Data Transformation

Relational database management systems serve a unique purpose in the world of technology. Databases of all types, from Microsoft Access to large mainframe systems, primarily function to store and manipulate data. Today's databases, particularly Oracle, are capable of storing massive amounts of information. However, storing data in a database is only part of the equation. If the end user cannot see this data presented in a meaningful way, the data loses much of its usefulness. This is why the SQL language is so important. SQL is the primary tool that the relational databases use to extract and present data. SQL makes it possible to sift through enormous amounts of data and highlight exactly the piece of data that is useful to the user. Doing so often involves analysis of some kind. **Data analysis** is the process of taking data and performing the necessary operations to transform it into a form that is relevant to solving problems. The subjects in this book cover the necessary techniques to enable SQL programmers to do analysis of this type. This chapter examines the last of the major clauses used in `SELECT` statements that allow for data analysis.

In this chapter, we shall:

- Examine the concept of grouping data
- Understand the syntax of the `GROUP BY` clause
- Study the concept of row group exclusion
- Explore the syntax of the `HAVING` clause
- Examine the types of aggregate functions available in SQL

Understanding multi-row functions

In the previous chapter, we examined the concept of a function. Our examples used functions that returned a value for every row used by the function. Many functions, however, take numerous values, such as all the values in a particular column, and return a single value. We call such functions multi-row functions.

Examining the principles of grouping data

In relational databases, all data is structured in some way. We've discussed relational concepts and noted the way that inter-table relationships contribute to this structure. Data is also structured within the table itself. Column values represent a certain attribute of the table. Often, part of the task of analysis is the grouping of pieces of data with a distinct attribute. Doing so usually involves the use of a function. We have seen single-row functions in the previous chapter that return one value for each row of input. The functions used in grouping data are **multi-row functions**: functions that take multiple rows as input and return a single, aggregated value. For this reason, multi-row functions are also referred to as **aggregate functions**. In this section, we will look at multi-row functions and later incorporate them with grouping.

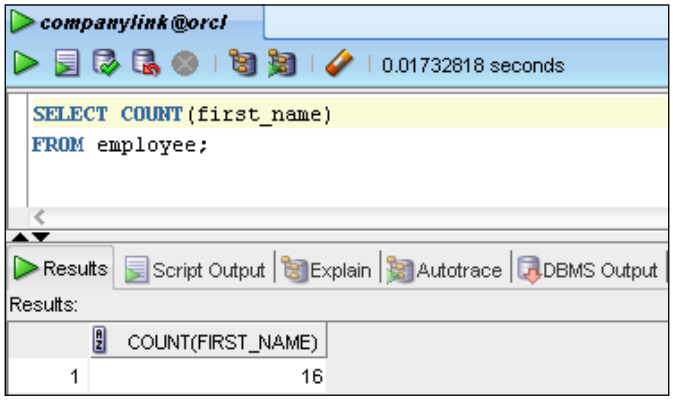
Using multi-row functions in SQL

As previously stated, multi-row functions take multiple rows as input and return a single value. Operations such as case conversion, done with single-row functions, obviously must take and convert rows individually, since the conversion occurs on every value. Multi-row functions are used when a different type of operation is needed, such as finding the minimum or maximum value from a list of values. A list of the common aggregate functions that we will examine is shown as follows:

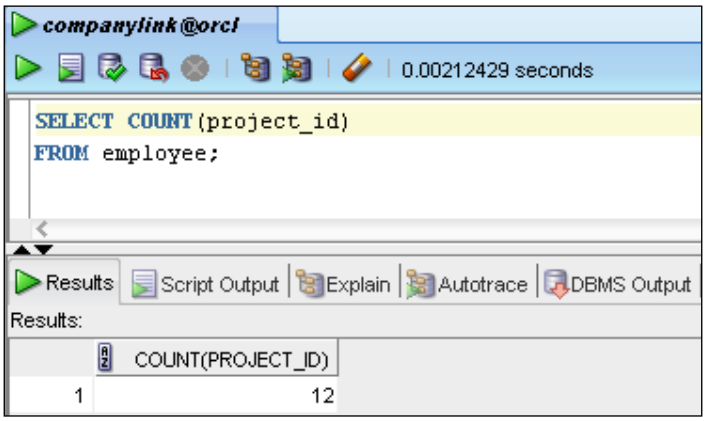
- `COUNT()`
- `MIN()`
- `MAX()`
- `SUM()`
- `AVG()`
- `STDDEV()`
- `VARIANCE()`

COUNT()

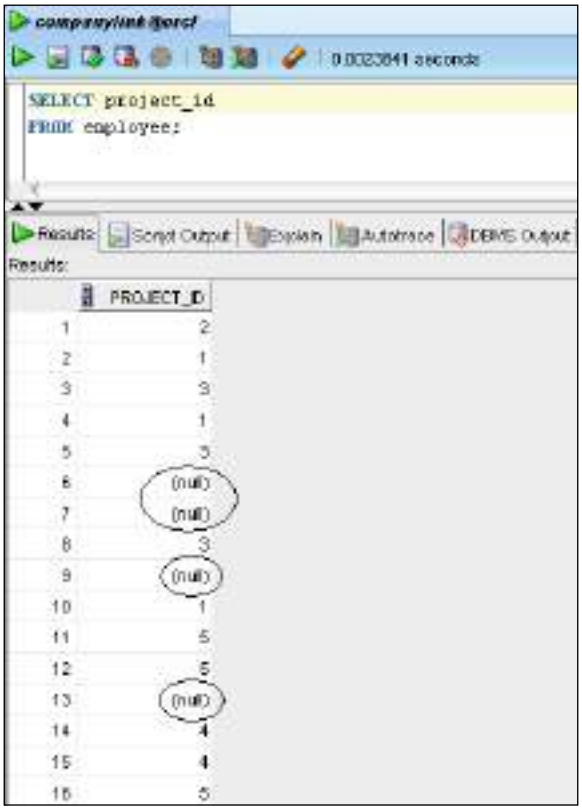
One of the simplest multi-row functions available to us is the `COUNT()` function. With it, we provide a column as an argument and return a count of the number of rows that were inputted. An example is shown in the following screenshot:



The `COUNT()` function, a multi-row or aggregate function, takes the `first_name` column as input and returns a count of the number of values for `first_name` that exist. In our `employee` table, there are 16 rows, and each row has a value for `first_name`, so a count value of 16 is returned. When using `COUNT()`, it is important to remember that the presence of nulls will affect the value returned. `COUNT()` will only count the values that are not `NULL`. Notice the effect that null values have in the following example:

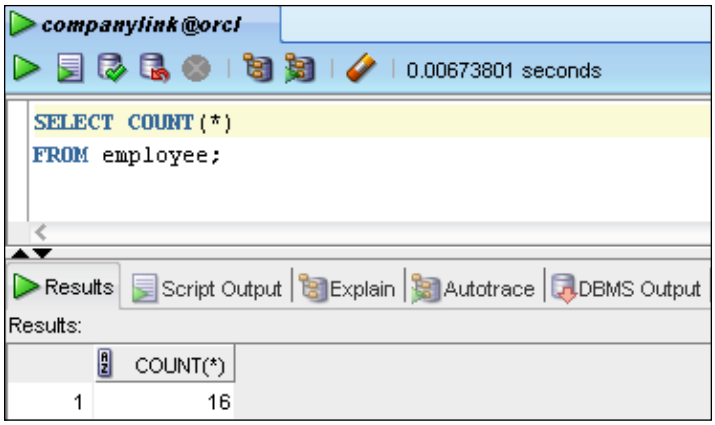


Here, a value of 12 is returned for our count. However, there are 16 rows in the table. Why, then, does a count of `project_id` values only yield a value of 12? To answer this, let's look at all the row values that are being passed as arguments. In the next example, we execute the previous statement without the `COUNT()` function. It returns all the values for `project_id`.



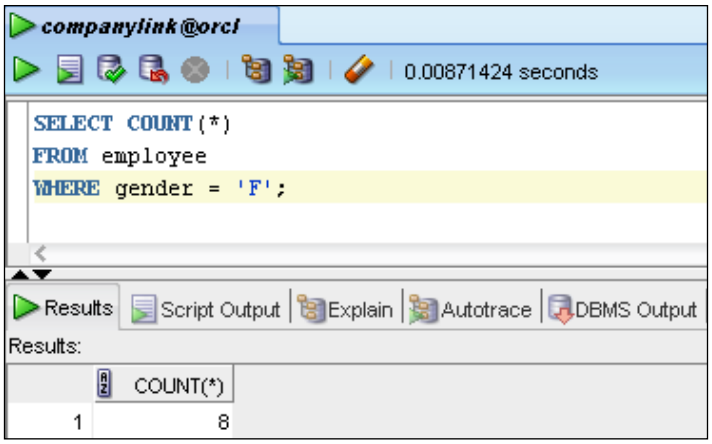
We see here that four of the 16 values for `project_id` in our `employee` table have a value of `NULL`. Since the `COUNT()` function only counts true values, NULLs are ignored. Subtracting these four values from the count, we see that 12 values are returned.

It is very common for SQL developers and database administrators to use the `COUNT()` function to get an overall count of the number of rows in a table. When we want to accomplish this, we pass an asterisk, `*`, as the function's argument. This is similar to using the asterisk with the `SELECT` clause to display all the rows in a table.



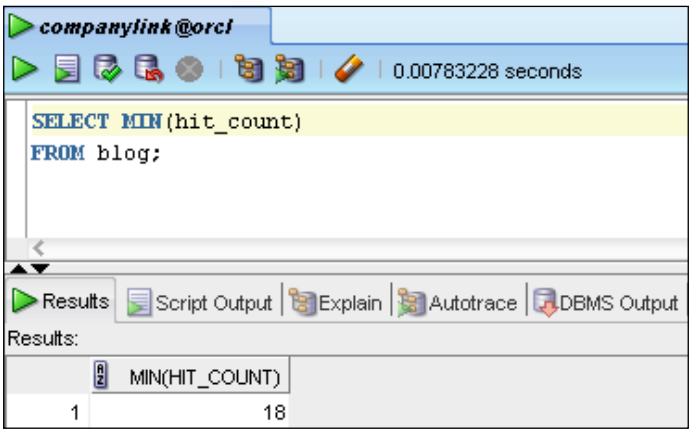
Note that `COUNT(*)` will count the rows in a table irrespective of the presence of null values. Even if the table had rows with nothing but null values, `COUNT(*)` will still return the actual number of rows in the table.

`COUNT(*)` is often used in conjunction with a `WHERE` clause to yield a count of the number of rows that meet a certain criteria. Say, for example, that we wanted to know how many employees in the `Companylink` database were female. We could use `COUNT(*)`, as shown in the following example, to find out. We could translate this statement as, *How many Companylink employees are female?*

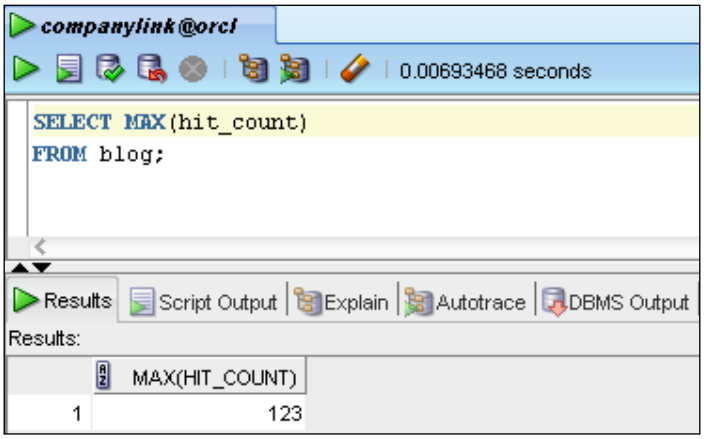


MIN() and MAX()

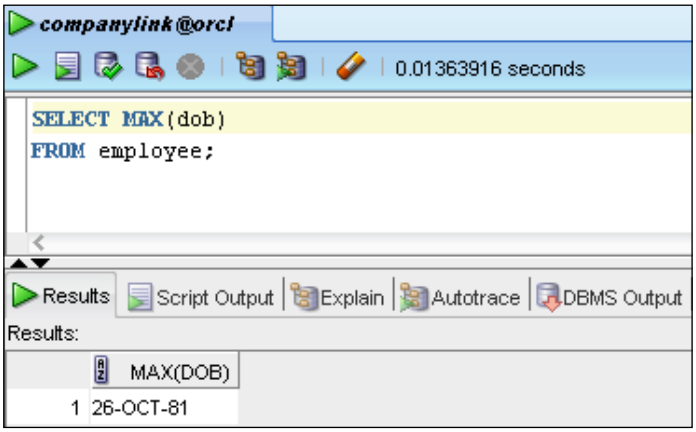
The `MIN()` and `MAX()` functions take a set of values, specified usually by a column, and return the minimum value or maximum value in the set, respectively. Let's say we want to use SQL to answer a simple question—*what is the fewest number of hits to any Companylink blog?* The solution is to use a function that can read through the list of hit statistics for blogs (found in the `hit_count` column in the `blog` table), compare them, and return the lowest value. This can be accomplished using the `MIN()` function as shown in the next example:



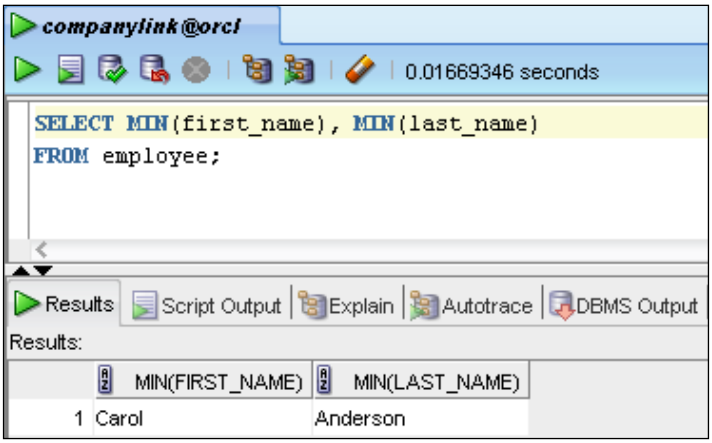
The `MAX()` function is the complement to `MIN()`. We use it to find the greatest in a set of values. Let's reverse our previous question—*What is the largest number of hits to any Companylink blog?* We can use the `MAX()` function as shown in the following screenshot:



While `MIN()` and `MAX()` are obviously most often used with numeric values, they will function with other datatypes as well. Say we want to find the most recent birthdate of any of our *Companylink* employees. We can pass the values for birthdates found in the `dob` column in the `employee` table, which has a datatype of `DATE`, to the `MAX()` function. It then returns the most recent date, shown as follows:



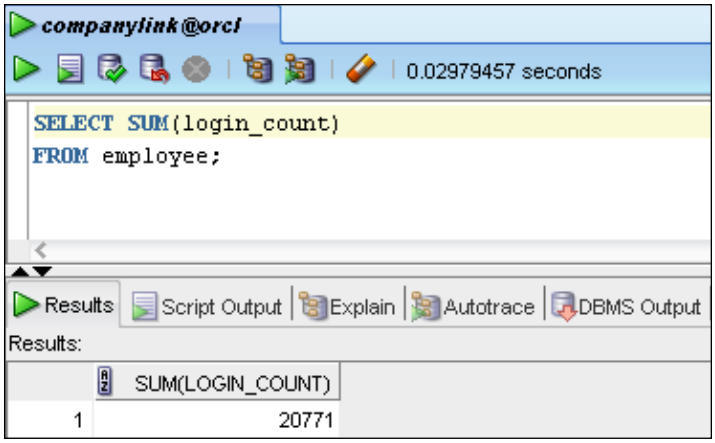
Even less common is the use of `MIN()` and `MAX()` with character string datatypes such as `VARCHAR2` and `CHAR`. However, it can be done, as shown in the following query:



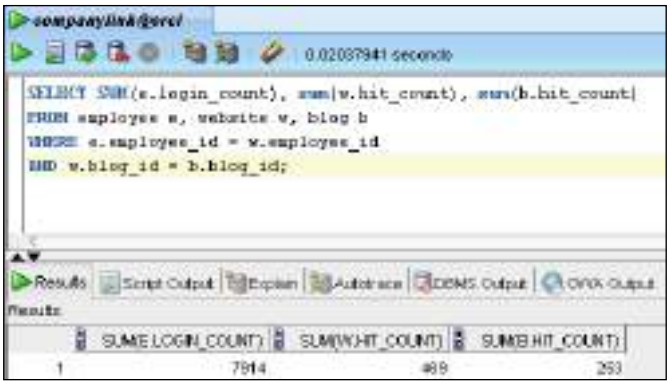
The `MIN()` function takes values from the `first_name` column of the `employee` table and returns the lowest value. But what constitutes the lowest value in a set of string literals? As we mentioned in *Chapter 3*, *Using Conditional Statements*, when referring to string values, numeric functions such as `MIN()` and `MAX()` interpret strings by their ASCII value. The value **Carol** and **Anderson** are returned since *C* and *A* are the lowest ASCII values for the first character found in the `first_name` and `last_name` column, respectively. When interpreting which character value is *higher* than another, it is often easier to think in terms of a character's place within the alphabet than its place within the ASCII table. 'A' is *lower* than 'B', since 'A' comes before 'B' in the alphabet.

SUM()

One of the most common operations needed in processing numeric data is the ability to calculate the sum of a set of values. It is so common that many commercial spreadsheet applications have a shortcut button to make it simple. In SQL, this operation is performed using the `SUM()` function. With `SUM()`, we pass in a set of numeric values and it returns their arithmetic sum. An example that shows the use of `SUM()` is shown in the next example. It could be read as, *Display the total amount of logins for all Companylink employees*.



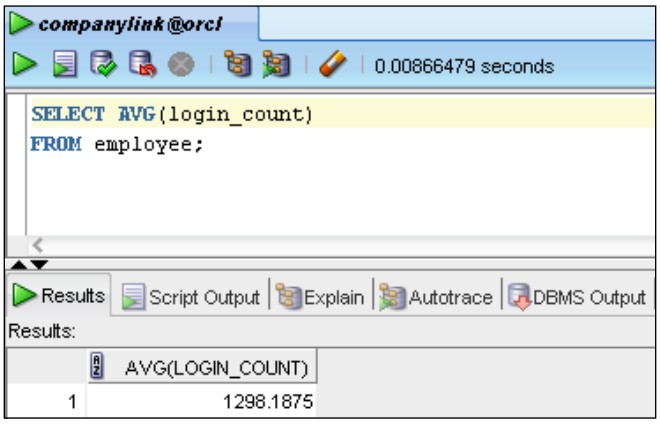
We can use what we have already learned about table joins to enable us to pull summed data from multiple tables and display it together, as we can see in the following screenshot:



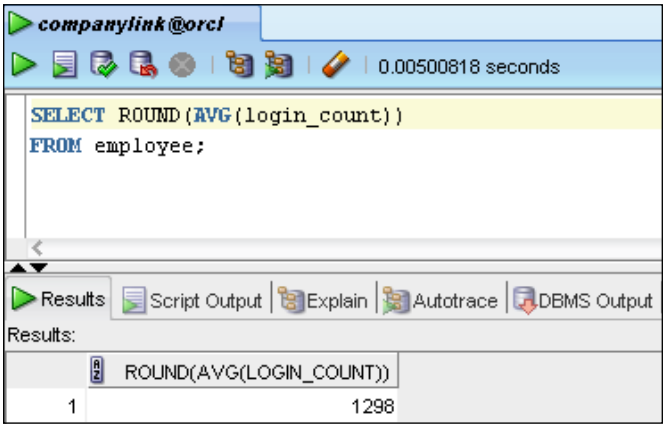
Here, we join three tables—`employee`, `website`, and `blog`—and use them to produce a summed value from each table—`login_count` from `employee`, `hit_count` from `website`, and `hit_count` from `blog`. Notice that since there are employees who don't have websites, our `login_count` in this query is significantly lower than the value from the previous query. Because we're joining the tables, only the `login_count` values from `employee` that have a matching website are counted. In short, we could phrase this command in real language as, *Display the total login and hit counts for employees with websites and blogs*.

AVG()

Mathematical averages are an integral component of any computing system. Averages can be used for diverse purposes: from calculating the average salary of a group of employees to computing the average distance from the sun of planetary bodies. Using the `AVG()` function, we pass in multiple values and return the arithmetic average. In the following example, we show `AVG()` being used to calculate the average number of logins from our *Companylink* employees.

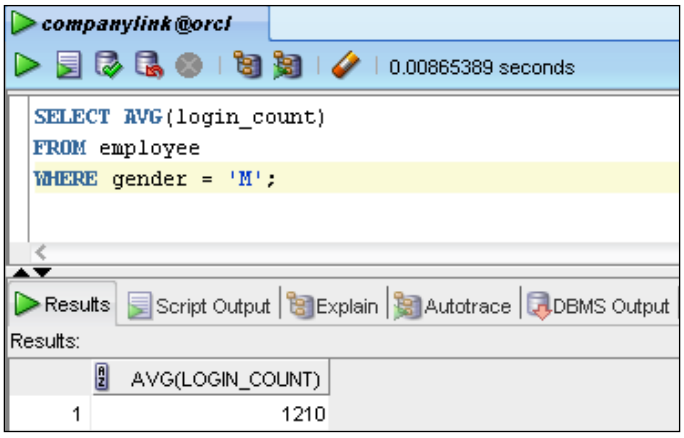


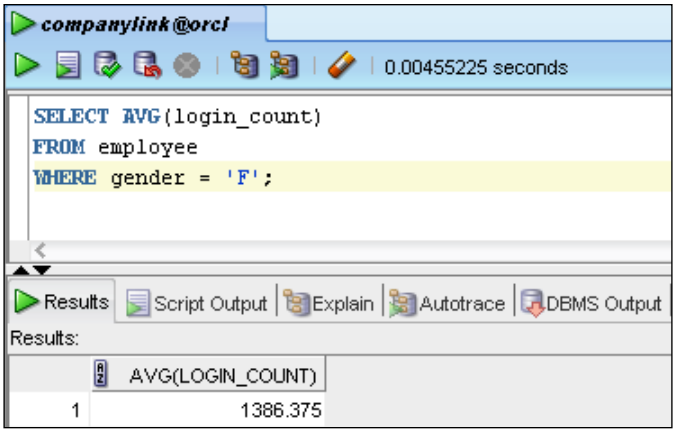
In this example, each value for `login_count` in the `employee` table is passed to the `AVG()` function. As with any average, the values are summed and divided by the number of input values, resulting in a mathematical average, in our case, with decimals. Assuming we want an integer value for our average number of logins, we could rewrite the previous statement using a single-row function, `ROUND()`, with the `AVG()` function nested within.



Grouping data

Clearly, the ability to do sums and averages with large amounts of business data is important. However, our previous examples have one significant limitation—the aggregated function operates on the entire table. Thus, we can see aggregation at one level, but if we want a more granular look at the data, we are forced to do a high degree of manual work breaking the data apart. For instance, we've seen an example of finding the average number of logins for all employees using the `AVG()` function. But, what if we wanted to see those averages on a different level? Say, the average number of logins differentiated between male and female employees? One way to do this is by utilizing two separate queries, using a `WHERE` clause to differentiate them. Examples of this are shown in the next two screenshots:





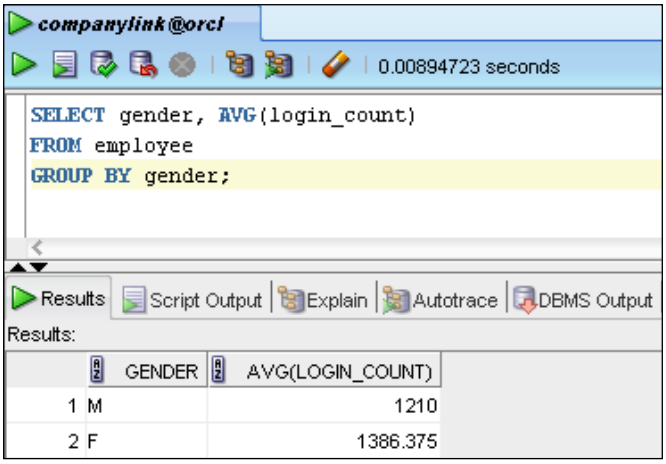
Thus, we see the average logins by employees, differentiated by gender. While this is serviceable in this example, what if our criteria for grouping was more complex? Let's say, for example, that our database serves a large company with thousands of employees, and we are required to find the average salary. But, the average must be computed based on one of hundreds of labor categories. To use the method mentioned previously, we would require hundreds of queries with `WHERE` clauses and is thus not practical. Fortunately, in SQL, we can accomplish such a task using the `GROUP BY` clause. With this clause, we can instruct Oracle to display aggregated functions in a grouping of our choosing.

Grouping data with GROUP BY

The syntax of the `GROUP BY` clause as it belongs within a `SELECT` statement is shown as follows:

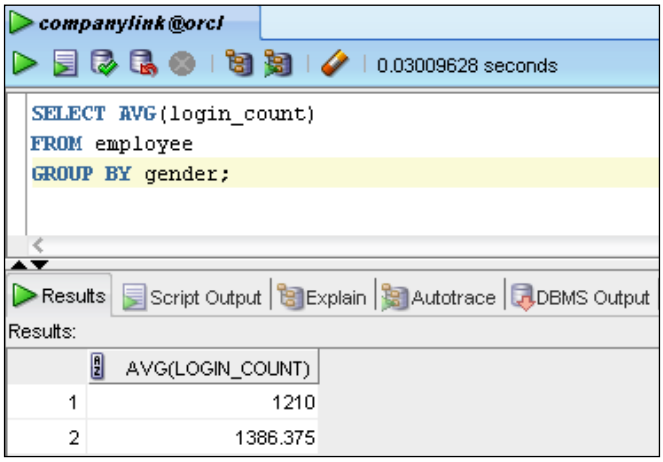
```
SELECT {column, column, …}
FROM {table}
WHERE {expression}
GROUP BY {expression}
ORDER BY {column};
```

Notice that the `GROUP BY` clause falls between the `WHERE` and `ORDER BY` clauses. Since both of these clauses are optional, it is possible to construct a statement that includes only a `SELECT`, `FROM`, and `GROUP BY` clause. The `GROUP BY` clause is paired with an expression, usually a column name, and also typically makes use of an aggregate function. A simple example is shown as follows:



Notice here that we have selected data from two columns: **GENDER** and **LOGIN_COUNT**. The **GENDER** column is displayed unmodified, but the values from `login_count` are passed to the `AVG()` function. The `GROUP BY` clause instructs Oracle to average the `login_count` based on the values in the `gender` column. Since `gender` holds two distinct values, **M** and **F**, two groupings and only two groupings are returned. If there were three distinct values in the `gender` column, three groupings with their respective averages would be returned.

Although we display the **GENDER** column here for clarity, it is not required. We can issue the same statement without selecting the **GENDER** column provided that we still use it to do the grouping. The revised example of this is shown in the following screenshot. The values returned for the averages are identical since the grouping conditions are the same.

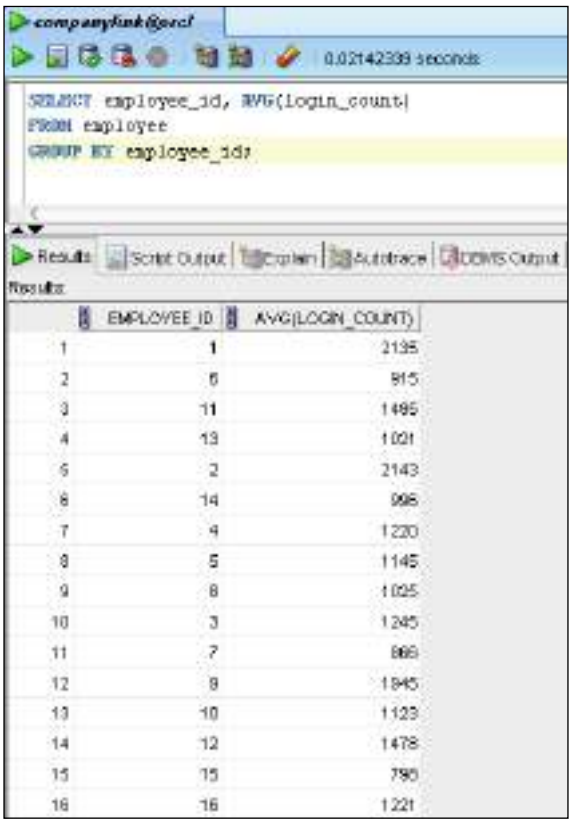


SQL in the real world

When learning SQL, it is a common misconception that one must always include every column that is used in any clause within the statement in your `SELECT` clause. While doing so is often more clear, it is not always necessary. For instance, you can use the `project_id` column in a `WHERE` clause or an `ORDER BY` clause without displaying it in the `SELECT` clause. Remember that each of the clauses in SQL do distinct operations that are not always dependent on each other.

Avoiding pitfalls when using GROUP BY

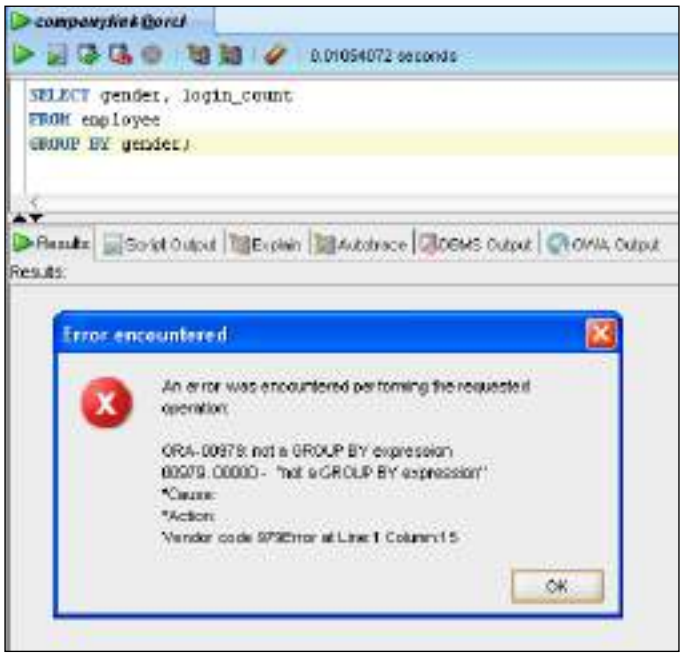
When using `GROUP BY`, it is important to include two crucial parts. First, we must group by a proper column. By *proper*, we mean that the column used in the `GROUP BY` expression must actually group the data in a way that is meaningful with respect to the multi-row function used. For example, look at the following query:



Here, we execute a statement identical to the previous one with one exception—the statement groups data based on `employee_id` instead of `gender`. No errors are returned, so why is this an improper use of the `GROUP BY` clause? Notice the data returned. When we grouped data using the `gender` column, there were only two distinct values on which to group, so only two groupings and their averages were returned. In this example, we are grouping based on **EMPLOYEE_ID**. 16 rows are returned, which is the number of distinct values for `employee_id` in the `employee` table. However, there are only 16 rows in the entire table. What has happened is that

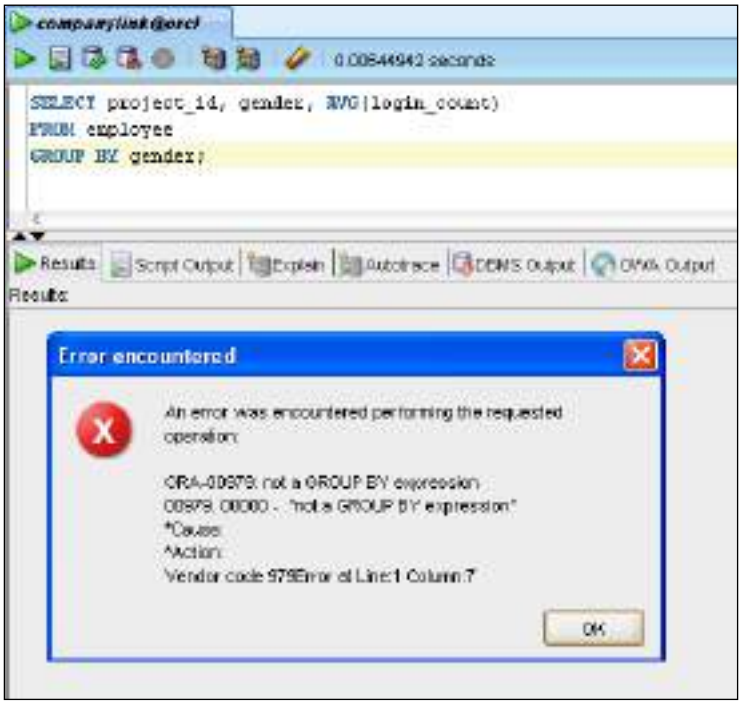
no grouping has taken place, since every distinct value is represented with its own row. Each `employee_id` is displayed along with an average of one value for each row. Since the average of one value is simply the value itself, the data shown is the same as if we had selected the `employee_id` and `login_count` values by themselves without a `GROUP BY` clause or `AVG()` function. In short, the column we use for grouping must group the rows returned in some meaningful manner.

The second crucial piece that must be remembered when using a `GROUP BY` clause is the aggregate function itself. In most cases, a `GROUP BY` is not used without a corresponding multi-row function. Doing so will result in an **ORA-00979** error—**"not a GROUP BY expression."** Notice this error in the following screenshot:

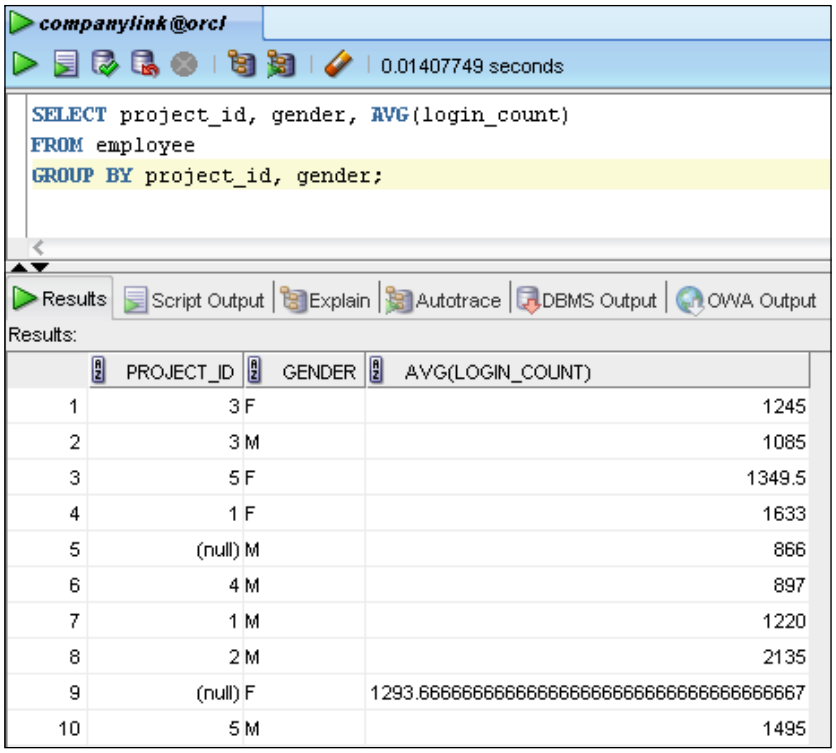


Here, we have the same statement we used previously except we've left off the `AVG()` function. We are grouping on `gender`, but there is no method specified for the grouping. As a result, no grouping can occur. When we group values, they must be grouped in a way that will return one row for each group.

Also note that when using the `GROUP BY` clause, we have restrictions from using other columns in the `SELECT` clause. In the next example, we again return to our original use of the `GROUP BY` clause, grouping `gender` on the average of `login_count`. However, we have made one small change—we've added an additional column, `project_id`, to our `SELECT` clause.



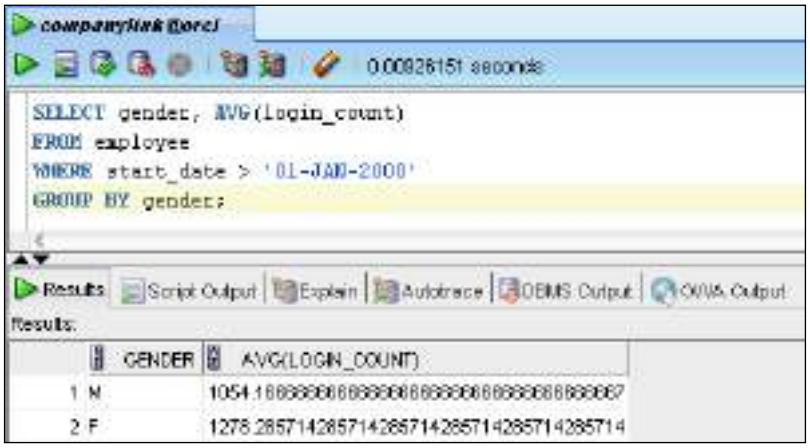
By adding the `project_id` column to our `SELECT` clause, we have asked SQL to do something that it cannot. When we group by `gender`, the result is two rows: one for **M** and one for **F**. If we add the requirement that the statement also display `project_id`, SQL cannot complete the request. There is more than one `project_id` associated with each of the values for `gender`, so which one of the `project_id` values should be shown? The commands in the statement give no direction on this, so the statement fails with an **ORA-00979** error. We can, however, correct this statement by grouping on more than one column. Notice the correction to the statement that we have made in the following query:



Here, we have instructed Oracle to retrieve three columns of data—`project_id`, `gender`, and the average of `login_count`. In order to return all three columns, we group by two different columns—`project_id` and `gender`. Notice the output. The data is grouped by **PROJECT_ID,** then by **GENDER**. If we notice the rows where the **PROJECT_ID** is **3**, we see two rows—one for **GENDER M** and another for **F**. These two rows correctly average the values for **LOGIN_COUNT** for each value of `gender` within the `project_id` grouping. Also notice the row where **PROJECT_ID** is **2**. The only employees with a **PROJECT_ID** of **2** are male, so only one grouping for a **PROJECT_ID** of **2** is shown. There are also rows where **PROJECT_ID** is **NULL**. Those rows are grouped with their two values for **GENDER** as well, although we can choose to exclude them, as we will see in the next section.

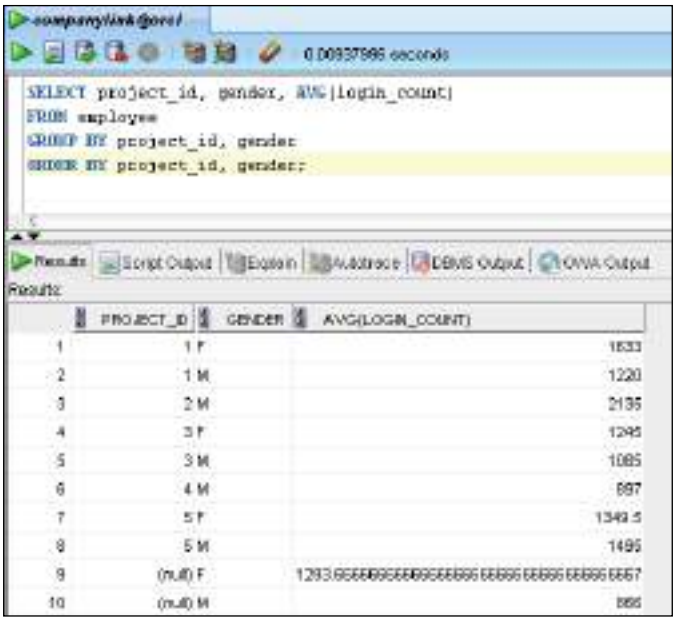
Extending the GROUP BY function

Using other clauses that we have learned in previous chapters, we can refine our use of `GROUP BY` to restrict and sort our rows. With the `WHERE` clause, we can restrict the values that are inputted into our aggregate function. Say, for example, that we wanted to again show the average number of logins for our *Companylink* employees and that we wanted to group them by `gender`. However, we only want to include employees who were hired after January 1, 2000. To do this, we restrict the rows inputted using a `WHERE` clause.

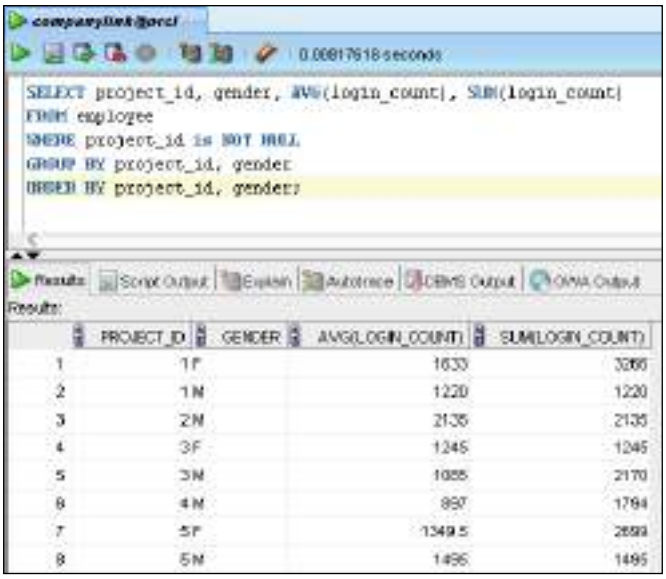


The resulting averaged values are different than if we had omitted the `WHERE` clause. The `WHERE` clause causes fewer rows to be passed to the `AVG()` function, thus changing the resulting averages.

Many of the examples thus far have resulted in output that is difficult to read. We can correct this using the `ORDER BY` clause. In our next query, we have rewritten the previous statement to sort the data by `project_id` and then by `gender`. The output shown is much clearer in terms of seeing the grouping.



We are also not restricted from using more than one aggregate function when using `GROUP BY`, provided that the functions can all logically coexist within the statement. Let's use all of what we've learned to write a statement that provides the data for a more complex request—*Display the average and total login_counts for all employees assigned to a project, grouped and sorted by their project and gender*.



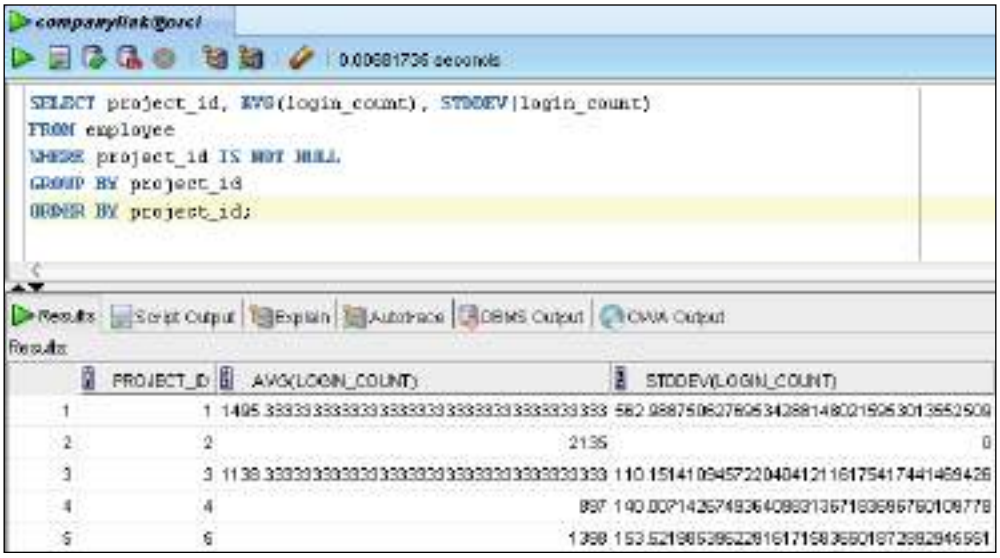
In this example, we display both the average and sum of login counts, then group and order them by `project_id` and `gender`. Since the requirement states that we only compute these values for employees who are assigned to a project, we use the `WHERE` clause to restrict input to only those employees whose `project_id` is not a `NULL` value.

Using statistical functions

Oracle's implementation of SQL comes with several functions designed for use with mathematical statistics and probability theory. While not every organization uses these, and they are not covered on the exam, we cover them briefly here as an example of the kind of advanced calculations that can be done with built-in SQL functions.

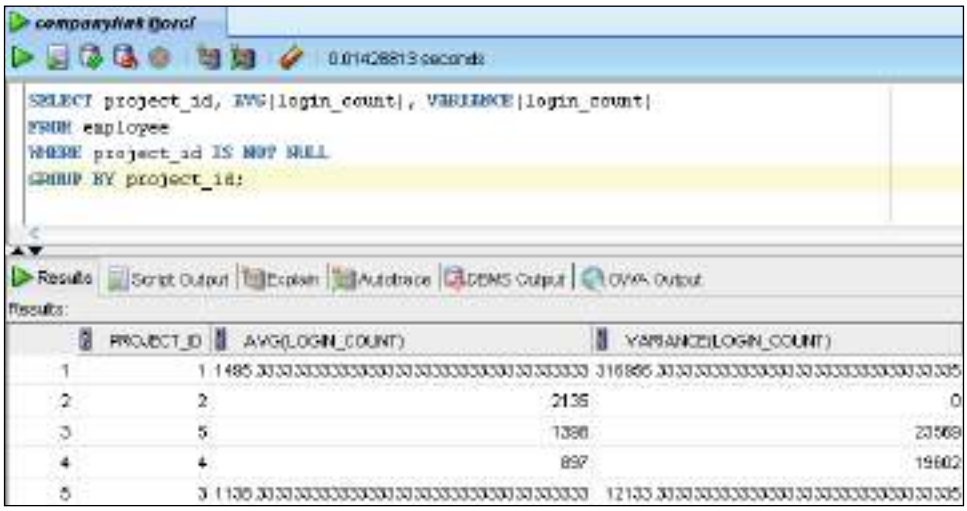
STDDEV()

In statistical theory, **standard deviation** represents the amount of variation that exists between a value within a set and an average of that same set of values. It measures how much a value *deviates* from the average. In the next example, we display the average `login_count` and standard deviation grouped by `project_id` using the `STDDEV()` function.



VARIANCE()

In probability theory, **variance** is the term used to describe how widely dispersed a group of values are. It is calculated against the minimum and maximum values of, in our case, a particular column. The larger the variance, the further the data is dispersed. In SQL, we use the `VARIANCE()` function to express this calculation, shown as follows:



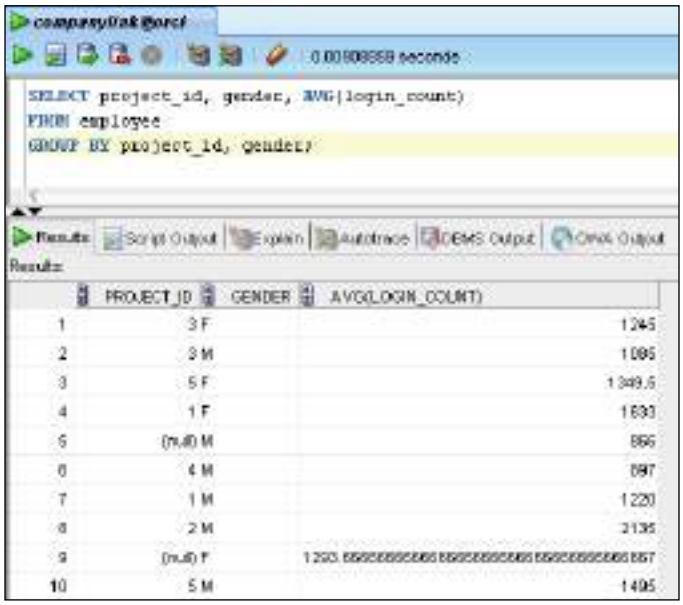
Performing row group exclusion with the HAVING clause

In the previous section, we examined using the `WHERE` clause to restrict the rows that are inputted into an aggregate function. This enables us to limit the data that is processed by the function. But, what if we need to limit the output of the groupings instead of the input? To do this, we use the `HAVING` clause with the basic syntax as shown:

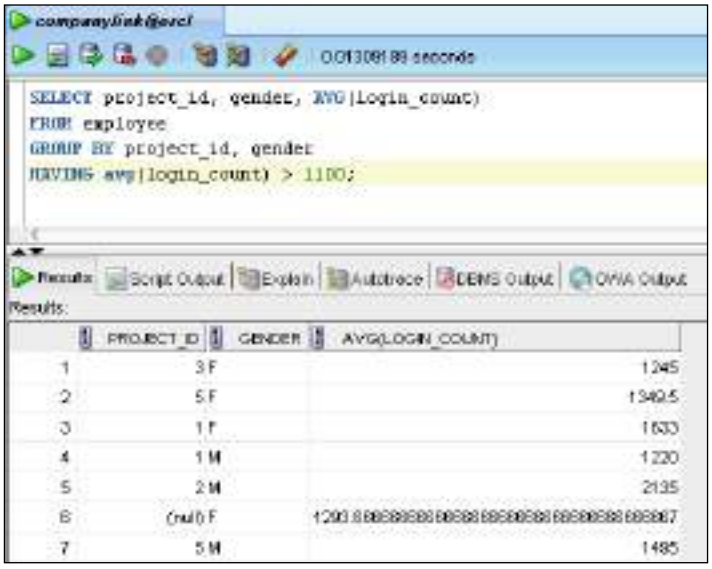
```
SELECT {column, column, …}
FROM {table}
GROUP BY {expression}
HAVING {expression};
```

Notice that the `HAVING` clause follows the `GROUP BY` in proper SQL syntax. While the two clauses can be reversed, it can make the statement difficult to read. While the `HAVING` clause is optional, it does require the presence of a `GROUP BY` in order to function properly. Failure to do so will result in an **ORA-00937** error—**"Not a single group function."**

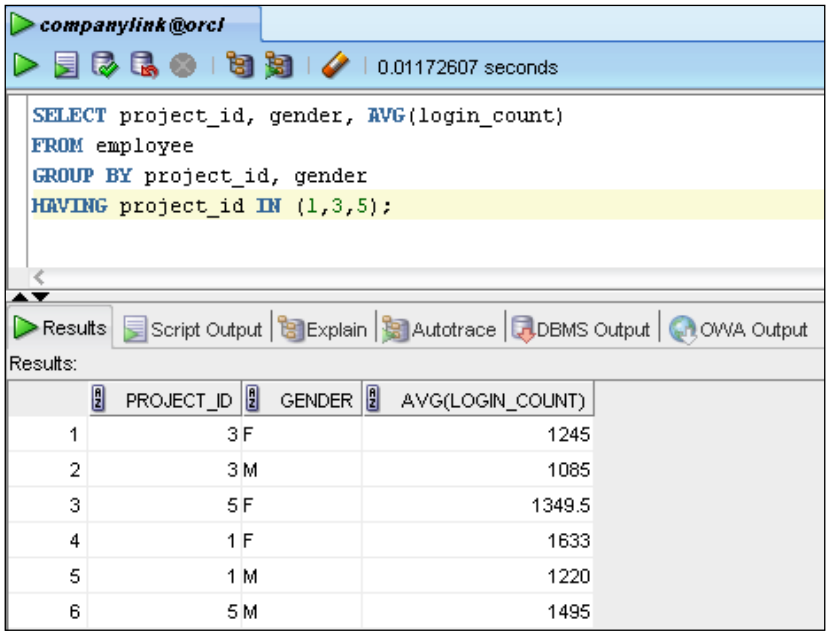
To see the `HAVING` clause at work, let's look at a previous query using a `GROUP BY`.



Here, we see the selected values shown in the groupings as requested. However, if we want to restrict the outputted row groups, say to display only groups with an average `login_count` that is greater than **1100**, we use the `HAVING` clause. We can see this group restriction in the following query:



Instead of displaying all of the groups, the `HAVING` clause excludes all row groups that do not meet the conditions specified in its expression—average `login_count` that is greater than **1100**. When using the `HAVING` clause, we are not limited to using the function, in this case `AVG()`, as our conditional expression. We can also use the same expressions in a `HAVING` clause to limit output as we do in a `WHERE` clause. The following example uses the `project_id` column to limit output and displays row groups meeting a condition specified by an `IN` operator:



	PROJECT_ID	GENDER	AVG(LOGIN_COUNT)
1	3	F	1245
2	3	M	1085
3	5	F	1349.5
4	1	F	1633
5	1	M	1220
6	5	M	1495

SQL in the real world

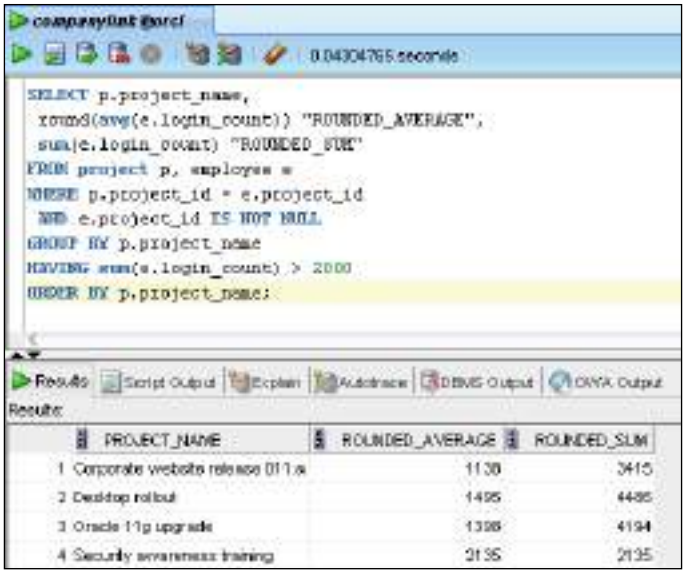
Because of the way that it restricts group output, the `HAVING` clause is often thought of as "a `WHERE` clause for `GROUP BY`". While this can make understanding the `HAVING` clause easier, it is important not to confuse the two. In statements such as these, the `WHERE` clause restricts the *input* to the function, while `HAVING` restricts its *output*.

Putting it all together

In *Chapter 2, SQL SELECT Statements*, we began with a simple `SELECT... FROM` statement. We then began adding clauses to accomplish different goals. With the addition of the `HAVING` clause, we come to the end of the essential clauses used in `SELECT` statements. While we will still explore other types of SQL statements and the techniques used with them, there are no more new clauses to add to the collection of SQL statements we use to query data. A proper syntax tree that combines all of the clauses for `SELECT` statements that we have seen throughout this book is shown as follows:

```
SELECT {column, column, …}
FROM {table}
WHERE {expression}
GROUP BY {expression}
HAVING {expression}
ORDER BY {column};
```

Nearly every SQL statement we can write involves taking these clauses and integrating them in ways that answer a particular question that we pose to the database. We close this chapter with an example that brings together each of the clauses we have seen. In it, we display login count information grouped together by project name. To do so, we join the `project` and `employee` tables in order to display the name of the project and limit the input rows to non-null values. We then group and sort them by `project_name` and limit the output rows to summed values greater than **2000**. We also round the average and use aliases to define our column headings.



Certification objectives covered

- Identify the available group functions
- Describe the use of group functions
- Group data by using the `GROUP BY` clause
- Include or exclude grouped rows by using the `HAVING` clause

Summary

In this chapter, we've added the final two clauses to our `SELECT` statements—`GROUP BY` and `HAVING`. We have used these two clauses to enable the grouping of rows for operations such as data analysis. We have examined the use of multi-row, or aggregate, functions to perform operations on a set of data in our `SELECT` statements. We then combined the grouping clauses and multi-row functions to compute group operations and display them in their respective groups. Finally, we brought all of our SQL clauses together to perform complex operations.

Although we may have come to the last of our SQL clauses, there is still more work to be done. In our next chapter, we'll learn a new SQL technique—subquerying. Using subqueries, we can nest queries inside of each other to combine data from tables in new ways. We'll follow that subject up by looking at set operations and set theory in Oracle.

Test your knowledge

1. Which of these statements describes the process of taking data and performing the necessary operations to transform it into a form that is relevant to solving problems?
    a. Data aggregation
    b. Data analysis
    c. Data subjugation
    d. Data elimination
2. The types of functions used in grouping data are called?
    a. Single-row functions
    b. Multi-row functions
    c. Restrictive functions
    d. Variance functions

3. Which of these statements describes how an aggregate function works?
   a. The function takes a single value and returns a single value
   b. The function takes a single value and returns multiple values
   c. The function takes multiple values and returns a single value
   d. The function takes multiple values and returns multiple values
4. Which of the following is NOT a multi-row function?
   a. COUNT()
   b. AVG()
   c. ROUND()
   d. STDDEV()
5. Given a table with 200 rows containing a column, SALARY, where 40 of the values for SALARY are NULL, what would be the result of issuing a COUNT(SALARY) in a statement?
   a. 200
   b. 160
   c. 40
   d. NULL
6. Given a table with 200 rows containing a column, SALARY, where 40 of the values for SALARY are NULL, what would be the result of issuing a COUNT(*) in a statement?
   a. 200
   b. 160
   c. 40
   d. NULL
7. Refer to the data in the website and blog tables in your Companylink database. Which of the following statements regarding MIN() and MAX() is true?
   a. MIN(hit_count) for website is greater than MIN(hit_count) for blog
   b. MIN(hit_count) for website is greater than MAX(hit_count) for blog
   c. MAX(hit_count) for website is less than MIN(hit_count) for blog
   d. MAX(hit_count) for website is greater than MAX(hit_count) for blog

8. Which of these statements would NOT result in an error?
   a. SELECT sum(*) FROM website;
   b. SELECT sum(website_desc) FROM website;
   c. SELECT sum(website_url) FROM website;
   d. SELECT sum(hit_count) FROM website;

9. Which of the following statements could be used to compute the same value as the following statement?

```
SELECT avg(login_count) FROM employee;
```

   a. SELECT sum(login_count) / count(login_count) FROM employee;
   b. SELECT stddev(login_count) FROM employee;
   c. SELECT stddev(login_count) * variance(login_count) FROM employee;
   d. SELECT min(login_count) * max(login_count) FROM employee;

10. Which of the following GROUP BY statements is both logically and syntactically correct?
    a. SELECT project_id, sum(login_count)
       FROM employee
       GROUP BY employee_id;
    b. SELECT project_id, login_count
       FROM employee
       GROUP BY project_id;
    c. SELECT project_id, sum(login_count)
       FROM employee
       GROUP BY project_id;
    d. SELECT project_id, gender, sum(login_count)
       FROM employee
       GROUP BY project_id;

11. Which of the following statements containing GROUP BY is syntactically incorrect?
    a. SELECT gender, avg(login_count)
       FROM employee
       GROUP BY gender;

b. SELECT gender, avg(login_count)
FROM employee
GROUP BY gender
WHERE login_count > 800;

c. SELECT gender, avg(login_count)
FROM employee
WHERE login_count > 800
GROUP BY gender;

d. SELECT gender, avg(login_count)
FROM employee
GROUP BY gender
ORDER BY avg(login_count);

12. Which of the following statements would group its output by project_id and limit it to maximum values for login_count greater than 1000?

a. SELECT project_id, max(login_count)
FROM employee
GROUP BY project_id;

b. SELECT project_id, max(login_count)
FROM employee
GROUP BY project_id
HAVING max(login_count) > 1000;

c. SELECT project_id, max(login_count)
FROM employee
GROUP BY project_id
HAVING min(login_count) > 1000;

d. SELECT project_id, max(login_count)
FROM employee
GROUP BY project_id
HAVING project_id > 1000;

8
Combining Queries

In *Chapter 5*, *Combining Data from Multiple Tables*, we learned a powerful method of combining data from multiple tables in the form of table joins. These joins commonly retrieve data from two or more tables by linking them through a common column. However, it is also important to be able to combine data in other ways. What if you are required to combine the data from two tables that have no columns in common? What if you need to read a single table more than once, combining the results of each iteration? For problems such as these, we can make use of another technique—the **subquery**. Subqueries do not require any new clauses in order to function. Rather, they take the existing clauses you have already learned and use them in new ways.

In this chapter, we will cover the following topics:

- Examining the types of problems that can be solved with subqueries
- Using different types of subqueries
- Examining set theory
- Utilizing different types of set operators

Understanding the principles of subqueries

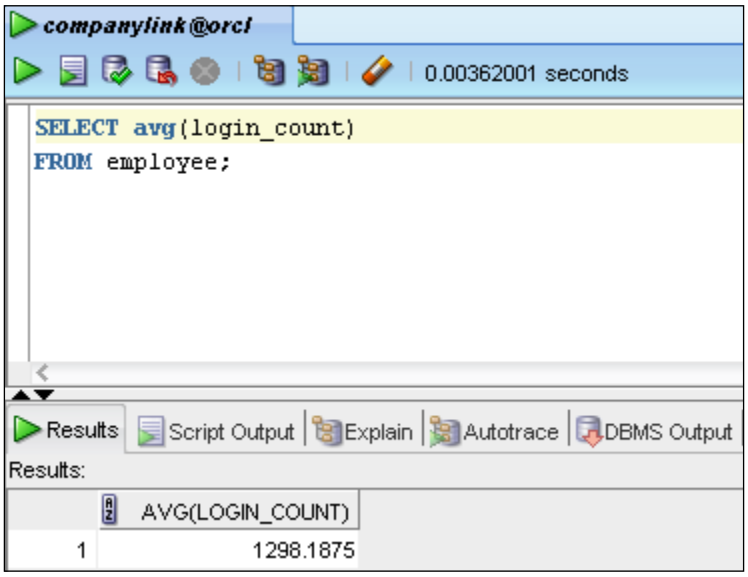
In *Chapter 5*, we first examined the topic of combining data from multiple tables. In that chapter, we used different kinds of join statements to combine data structures formed from the relationships between tables. In this chapter, we explore a new way of combining data from multiple tables.

Accessing data from multiple tables

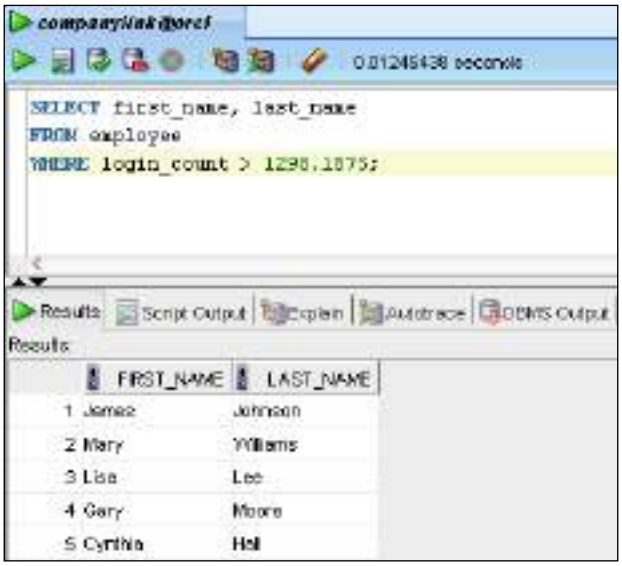
A subquery is simply a query that is nested inside another query. The nested query, sometimes referred to as an inner query, is evaluated first, and its resulting data set is passed back to the outer query and evaluated to completion. The result set obtained from the inner query can be a single value, multiple values, or even multiple columns. Subqueries can be used in many different clauses, as well. Whereas a join is useful when combining data from multiple tables, using an established relationship between the two tables. Subqueries can be effectively used in situations where the combined data has no direct relationship. We will examine each of these types of subqueries and their syntactical rules in this chapter.

Solving problems with subqueries

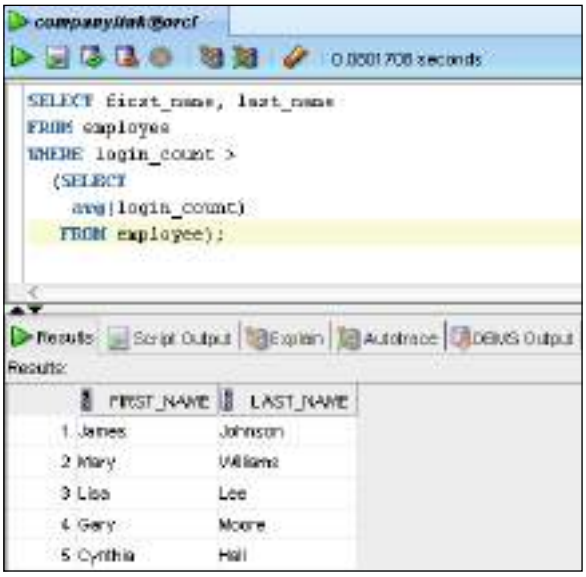
When administering a website like *Companylink*, factors such as page hit counts and numbers of logins are very important. The examples in this chapter focus on this type of data. Subqueries can often help SQL programmers to mine and aggregate such data to develop business intelligence. To get an idea of the kinds of problems that we can solve using subqueries, consider the following simple request: *Display the names of employees whose number of logins to Companylink exceed the average*. Using what we've learned so far, our only solution would be to use two queries, as shown in the next two screenshots. First, we use a multi-row function to find the average:



We next take that value and plug it in to the appropriate query for employee name information.



While using these two queries to find the requested information would suffice, a more direct method using a single statement is possible when we use a subquery. A subquery that retrieves such information is shown in the following screenshot:



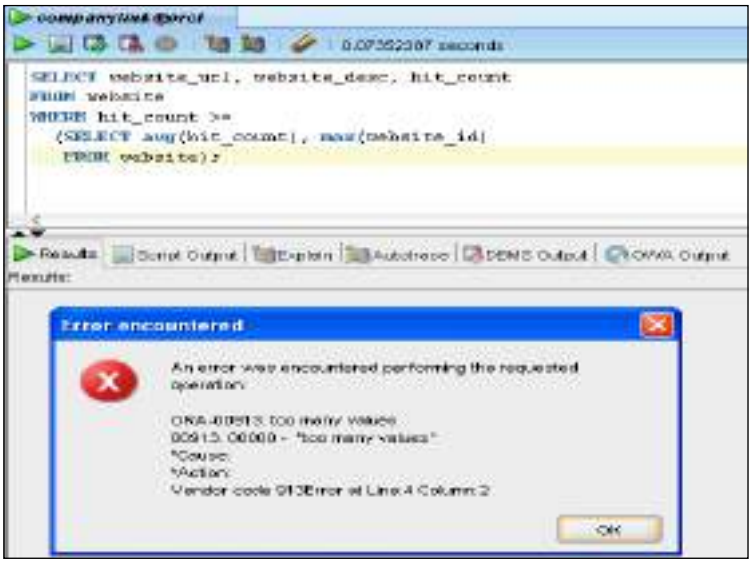
Here, rather than using a literal numeric value as a `WHERE` clause condition, we replace the condition with a subquery. The nested subquery evaluates to the same value we saw previously, `1298.1875`, and it becomes the condition in the `WHERE` clause. The outer query then returns the first and last names that meet those conditions. Also note that, syntactically, the `SELECT` statement that forms the inner subquery is enclosed in parentheses and is not followed by a semicolon. Our statement-terminating semicolon is placed at the end of the statement.

Examining different types of subqueries

In this section, we will learn how to examine the following types of subqueries:

Using scalar subqueries

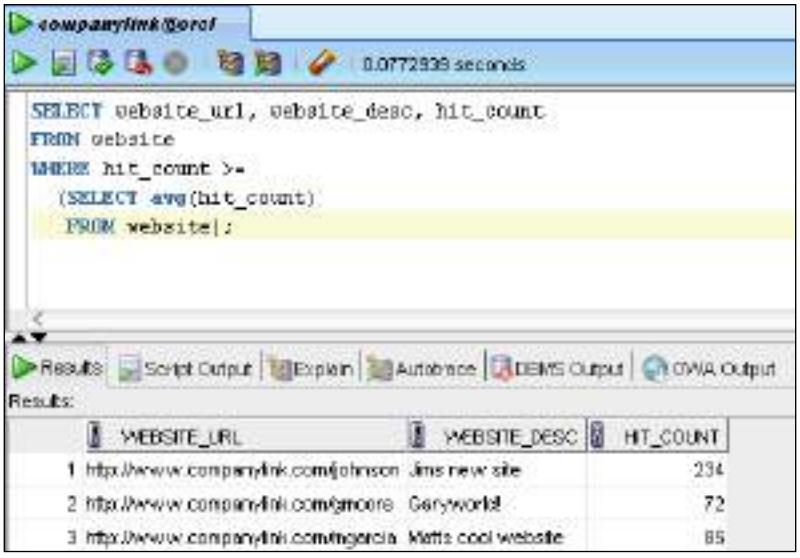
Subqueries such as the ones we've seen so far are known as **scalar** or **single-row subqueries** because the subquery returns a single value to the outer query. This single value replaces the condition in the `WHERE` clause. When writing statements that use subqueries, we say that certain types of queries expect either a single value or multiple values. The previous statement expects one value, as the `WHERE` clause can only be evaluated properly if one value is returned. For instance, let's say that we want a list of websites with the average highest hit counts on *Companylink* grouped by the website description. To attempt this, we write a statement using a subquery as shown:



When we run the statement, an error is returned. Even though the inner query is syntactically correct, it returns more values than can be evaluated by a single `WHERE` statement. Thus, an **ORA-01427** error, **single row subquery returns more than one row**, is returned. The `WHERE` clause expects a single value, but the subquery returns multiple values.

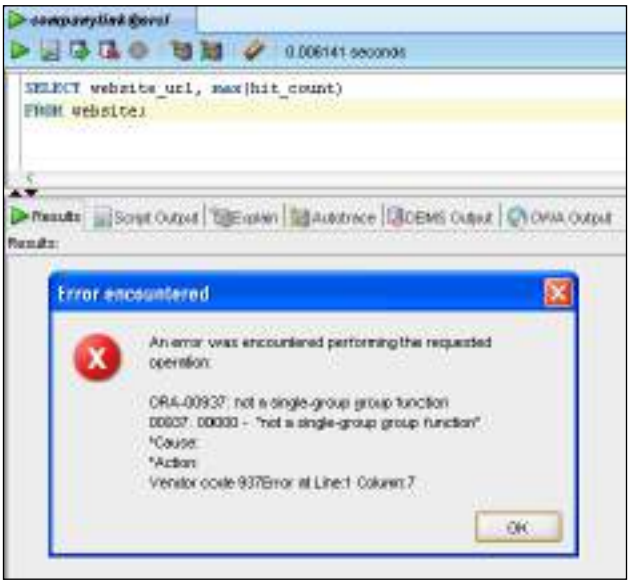
Using scalar subqueries with WHERE clauses

We can correct the previous incorrect statement by removing the grouping from the subquery, as we see in the next example:

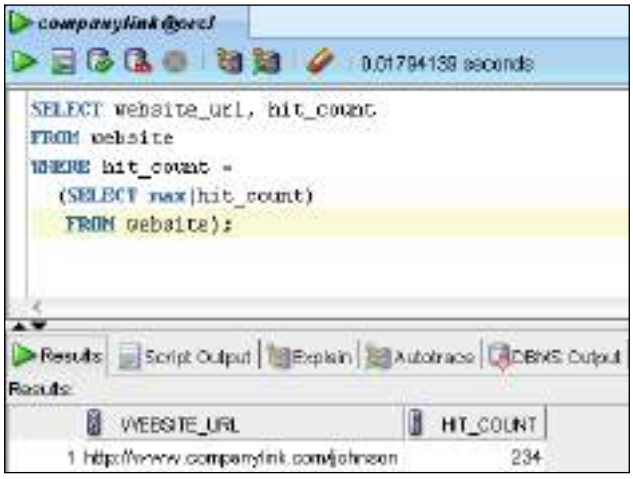


The majority of scalar subqueries are used as conditions for `WHERE` clauses. The types of operators used in these conditions are the same as those used in a typical `WHERE` clause. Conditions of equality or non-equality can be used, such as `<`, `>`, `<=`, `>=`, or `=`, provided that the subquery returns only one value. Say, for example, that we are asked to use SQL to find the website URL with the highest hit count for the `Companylink` application. Because our website table is small, we could simply select all the values in the table and then view them. But, if our table was larger, or our application needed to return only one row for the sake of display on a webpage, this would not be practical.

We might incorrectly attempt this using the statement shown by the following error:



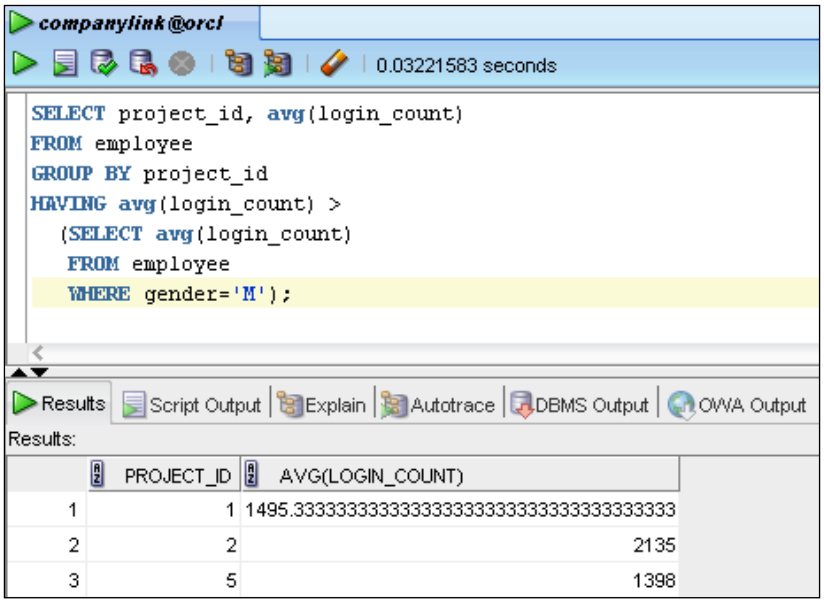
We learned in the previous chapter that we cannot successfully execute this statement as we cannot select a column and an aggregate of table rows without a `GROUP BY` clause. In this case, a `GROUP BY` clause would not return the maximum value that we seek. Grouping by `website_url` would return a rowset that includes every URL and its `hit_count`, not the maximum. To do this correctly, we would use a subquery, as shown in the following example. We might translate this query into real language as, *Find the website with the highest number of hits*.



This statement does not generate an error, as the outer query uses no aggregate function. The inner query uses the `MAX()` function, but requires no `GROUP BY` clause because no other column is selected. The result is the website URL with the highest number of hits.

Using scalar subqueries with HAVING clauses

We have already seen the use of subqueries with the `WHERE` clause, but subqueries can be used with many different clauses within an SQL statement. A scalar subquery can be used as the condition for a `HAVING` clause as well. Using subqueries with the `HAVING` clause can be useful when you want to group the output of a query by one column and then restrict its output based on a different column using a `HAVING` clause. For instance, say that we want to display the average login count grouped by `project_id`. But, we only want to display groups that have a higher than the average login count for male employees. The `HAVING` clause is useful here because it limits the group output. A subquery is appropriate as the condition because it determines the average logins for male employees. A statement that accomplishes this is shown in the following screenshot:



Subqueries used with `HAVING` clauses allow for many of the same types of conditions as `WHERE` clauses. The `IN` operator is not commonly used in scalar subqueries, as it usually refers to a set of values instead of a single value. The use of an `IN` operator to compare a single value is essentially the same as using an = condition. As we will see, the `IN` operator is much more common in multi-row subqueries.

Using scalar subqueries with SELECT clauses

Subqueries that return a single value can also be used in the `SELECT` clause of a SQL statement. While this may seem unusual, this technique can be useful for presenting a comparison value alongside a table column. Say we've been tasked with writing a report that compares each employee's login count to the average and displays this comparison in a way that is easy to read. We could embed a subquery within our `SELECT` statement to do this, as shown:

companylink@orcl

0.0466238 seconds

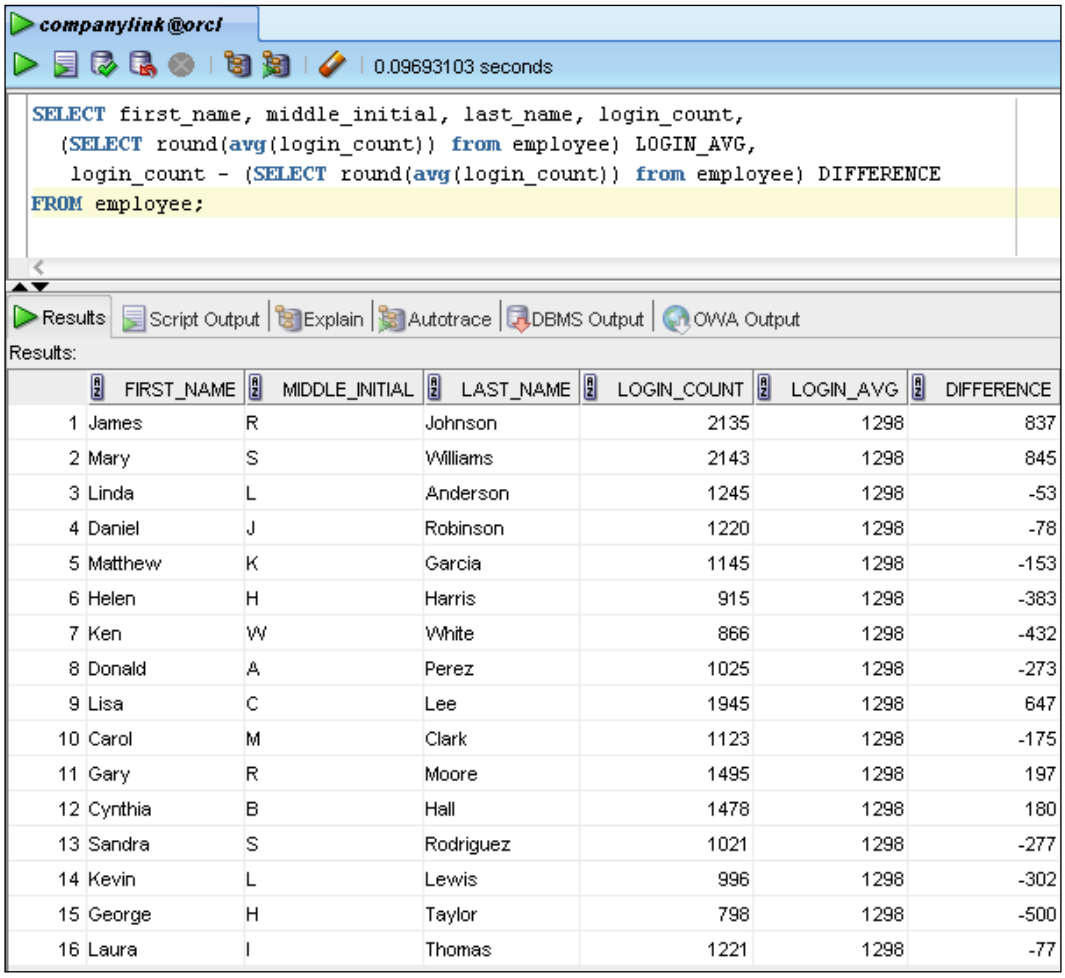
```
SELECT first_name, middle_initial, last_name, login_count,
  (SELECT round(avg(login_count)) from employee) LOGIN_AVG
FROM employee;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	LOGIN_COUNT	LOGIN_AVG
1	James	R	Johnson	2135	1298
2	Mary	S	Williams	2143	1298
3	Linda	L	Anderson	1245	1298
4	Daniel	J	Robinson	1220	1298
5	Matthew	K	Garcia	1145	1298
6	Helen	H	Harris	915	1298
7	Ken	W	White	866	1298
8	Donald	A	Perez	1025	1298
9	Lisa	C	Lee	1945	1298
10	Carol	M	Clark	1123	1298
11	Gary	R	Moore	1495	1298
12	Cynthia	B	Hall	1478	1298
13	Sandra	S	Rodriguez	1021	1298
14	Kevin	L	Lewis	996	1298
15	George	H	Taylor	798	1298
16	Laura	I	Thomas	1221	1298

Notice, syntactically, that the subquery follows a comma in the list of columns and precedes the `FROM` clause. For the purpose of clarity, we encapsulate the `AVG()` function within a `ROUND()` function to remove the decimal, and add a column alias, `LOGIN_AVG`. In this example, we are essentially using a subquery to create a column that doesn't exist in the table. Rather, it is a column that is derived from the data used for the purposes of comparison. We can also use derived columns such as these to do computation. With the addition of one line of SQL, we can add another derived column that shows the difference between the employee's login count and the rounded average. In the following example, we add a derived column with the heading `DIFFERENCE` that subtracts the rounded average from the employee's `login_count`. This shows positive values for employees with a higher than average login count and negative values for lower than average:



companylink@orcl

0.09693103 seconds

```
SELECT first_name, middle_initial, last_name, login_count,
  (SELECT round(avg(login_count)) from employee) LOGIN_AVG,
   login_count - (SELECT round(avg(login_count)) from employee) DIFFERENCE
FROM employee;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

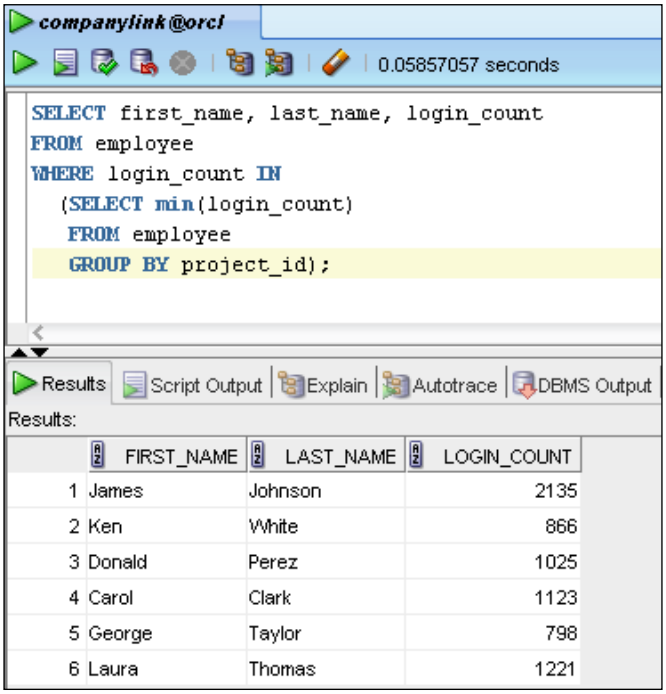
	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME	LOGIN_COUNT	LOGIN_AVG	DIFFERENCE
1	James	R	Johnson	2135	1298	837
2	Mary	S	Williams	2143	1298	845
3	Linda	L	Anderson	1245	1298	-53
4	Daniel	J	Robinson	1220	1298	-78
5	Matthew	K	Garcia	1145	1298	-153
6	Helen	H	Harris	915	1298	-383
7	Ken	W	White	866	1298	-432
8	Donald	A	Perez	1025	1298	-273
9	Lisa	C	Lee	1945	1298	647
10	Carol	M	Clark	1123	1298	-175
11	Gary	R	Moore	1495	1298	197
12	Cynthia	B	Hall	1478	1298	180
13	Sandra	S	Rodriguez	1021	1298	-277
14	Kevin	L	Lewis	996	1298	-302
15	George	H	Taylor	798	1298	-500
16	Laura	I	Thomas	1221	1298	-77

Processing multiple rows with multi-row subqueries

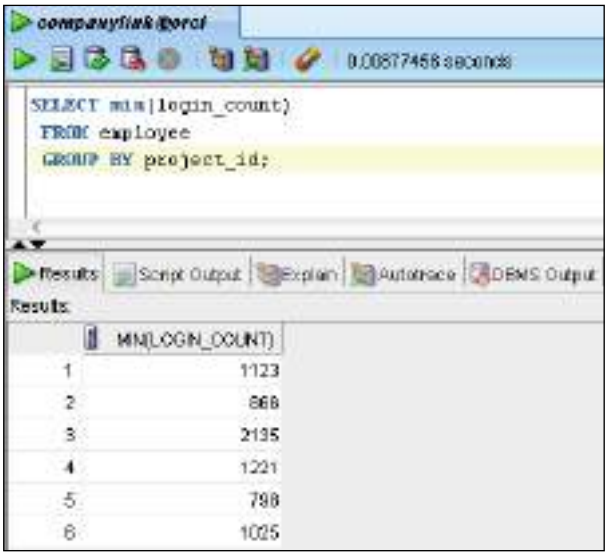
Unlike scalar, single-row queries, **multi-row subqueries** return multiple values to the outer condition. As such, they require different syntax and rules. As we have seen in previous examples, we cannot use the typical conditions of equivalence and non-equivalence with multi-row subqueries, as multiple values cannot be said to be equal to a single value. As we will see, multi-row subqueries are most commonly used in `WHERE` and `HAVING` clauses.

Using IN with multi-row subqueries

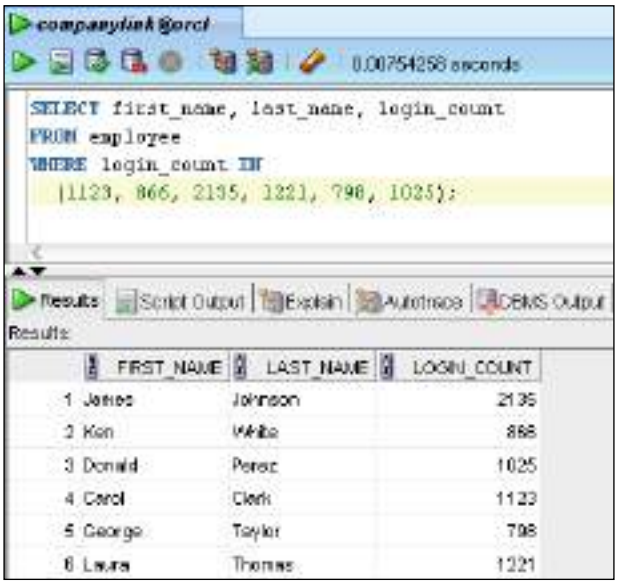
As multi-row subqueries return multiple values to the outer query, the operators that precede them must be capable of handling multiple values. As we stated before, operators that perform equivalence and non-equivalence functions are not capable of this. Thus, we must use multi-row operators such as `IN`. We have used the `IN` operator before with simple `WHERE` clauses in *Chapter 3*, *Conditional Row Retrieval and Sorting Data*. But, we can also use them in multi-row subqueries, as we can see in the following screenshot:



For simplicity, let's deconstruct this statement into its component clauses. First, let's look at the subquery and its output:



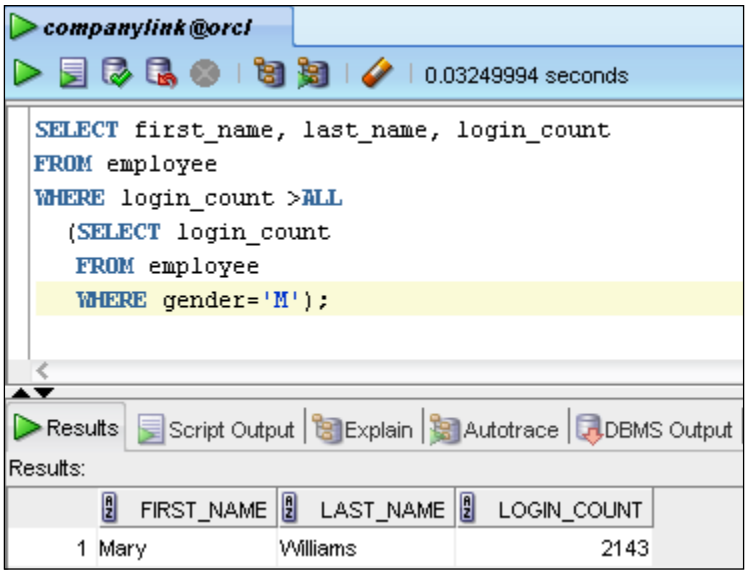
The subquery calculates the minimum value for `login_count` grouped by `project_id`. Thus, it displays the lowest employee login count for each project. The multiple values are passed to the outer query as shown in this example. In it, the outer query receives multiple values from the subquery.



Once the values from the subquery are received, the outer query processes them as if they were literal values. The outer query matches the `first_name`, `last_name`, and `login_count` to each of the minimum values found for each `project_id`. Thus, the full query satisfies the following request: *Display the name and login counts for each employee with the lowest login count for each project*.

Using ANY and ALL with multi-row subqueries

As we've stated previously, multi-row subqueries bring back a group of values to the outer query. While the `IN` operator evaluates each value in the group to check for equivalence, two special operators, `ANY` and `ALL`, evaluate the values as a whole. `ANY` and `ALL` treat the values returned by the subquery as a set instead of discrete values. `ANY` and `ALL` are unique to multi-row subqueries and are always paired with non-equivalence operators such as `<`, `>`, `<=`, and `>=`. The following screenshot demonstrates the use of the `ALL` operator:



We look first at the subquery, which returns all the `login_count` values for male employees. The values returned constitute the set of values that the outer query will compare against. Next, the `WHERE` clause used with `>ALL` in the outer query will return all values that are greater than every value in the set. In short, `>ALL` means that the returned values must be greater than all the values passed to the outer query. In our example, only values that are greater than all the `login_count` values for male employees will be returned, which naturally means that the values returned will be for female employees. Let's take a look at a similar statement using the `<ANY` operator:

companylink@orcl

0.00983198 seconds

```
SELECT first_name, last_name, login_count
FROM employee
WHERE login_count <ANY
  (SELECT login_count
   FROM employee
   WHERE gender='M');
```

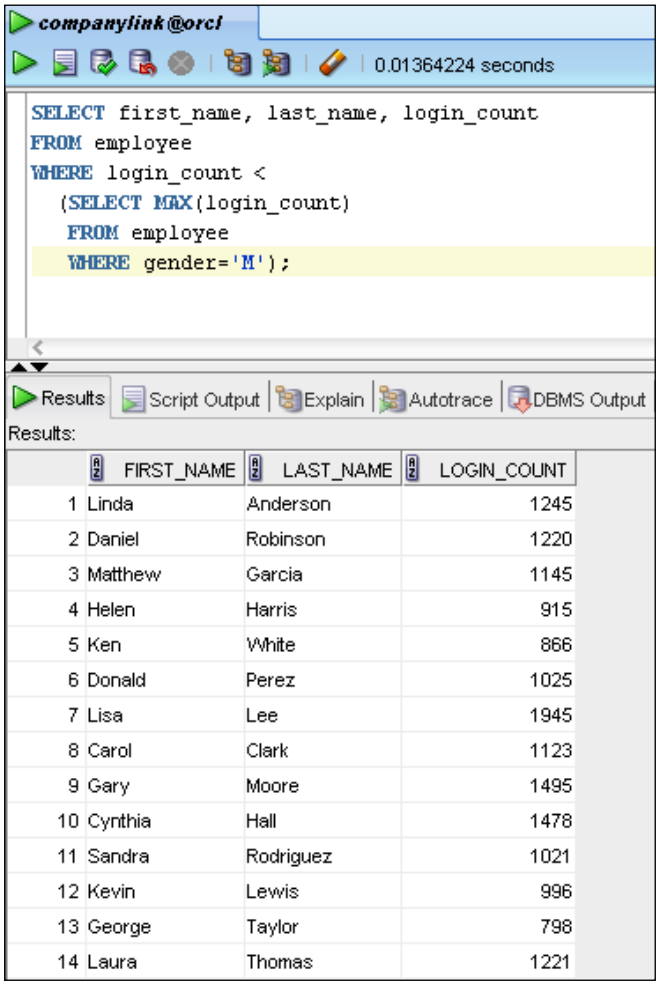
Results | Script Output | Explain | Autotrace | DBMS Output

Results:

	FIRST_NAME	LAST_NAME	LOGIN_COUNT
1	George	Taylor	798
2	Ken	White	866
3	Helen	Harris	915
4	Kevin	Lewis	996
5	Sandra	Rodriguez	1021
6	Donald	Perez	1025
7	Carol	Clark	1123
8	Matthew	Garcia	1145
9	Daniel	Robinson	1220
10	Laura	Thomas	1221
11	Linda	Anderson	1245
12	Cynthia	Hall	1478
13	Gary	Moore	1495
14	Lisa	Lee	1945

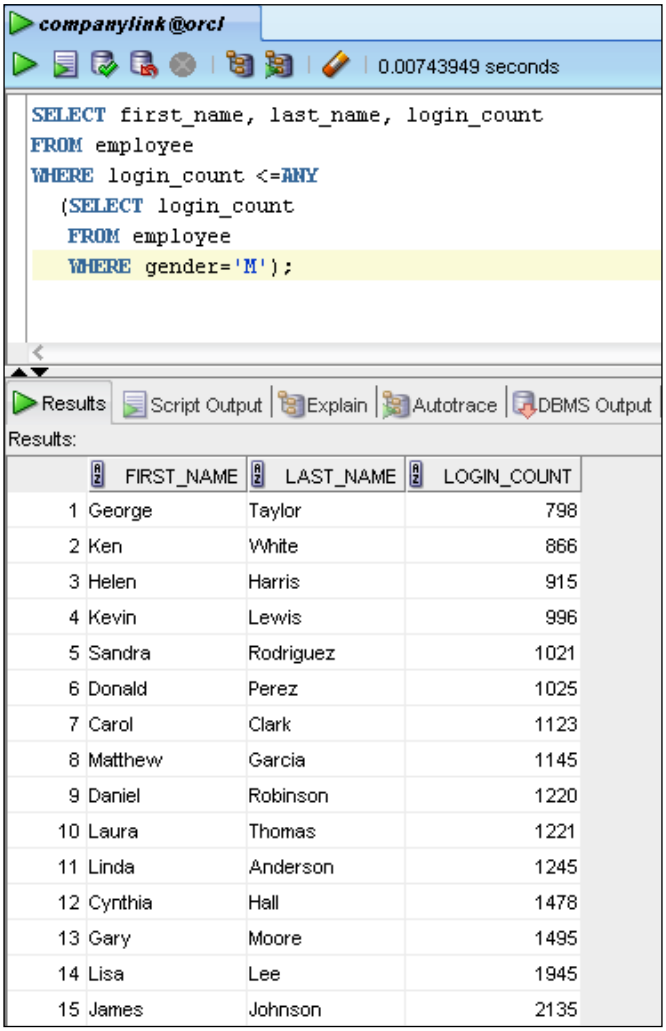
Again, our subquery returns login counts for male employees. However, this statement uses the `ANY` operator paired with a less than. The `<ANY` in a `WHERE` clause will return values less than the highest value in the set. The highest value in the subquery is `2135`, so the outer query returns all employee name and login count information for any employee with a `login_count` less than `2135`.

It might occur to you that as `<ANY` returns values less than the highest (or maximum) value in the subquery for comparison, we should be able to rewrite the previous query using the `MAX()` function instead. In this case, we certainly can, as shown in the following query. We remove the `ANY` operator and apply the `MAX()` function to the `login_count` in the subquery. The results are the same, although ordered differently:



	FIRST_NAME	LAST_NAME	LOGIN_COUNT
1	Linda	Anderson	1245
2	Daniel	Robinson	1220
3	Matthew	Garcia	1145
4	Helen	Harris	915
5	Ken	White	866
6	Donald	Perez	1025
7	Lisa	Lee	1945
8	Carol	Clark	1123
9	Gary	Moore	1495
10	Cynthia	Hall	1478
11	Sandra	Rodriguez	1021
12	Kevin	Lewis	996
13	George	Taylor	798
14	Laura	Thomas	1221

We can also use `ANY` and `ALL` with greater than or equal to and less than or equal to operators (`<=`, `>=`). These operators will work the same as the ones we've seen so far, with the only difference being that they will return any values equal to the lowest or highest. In the next example, we rewrite the previous statement that used `<ANY` to use `<=ANY`. The query returns one more value than the original—the lowest value in the subquery:



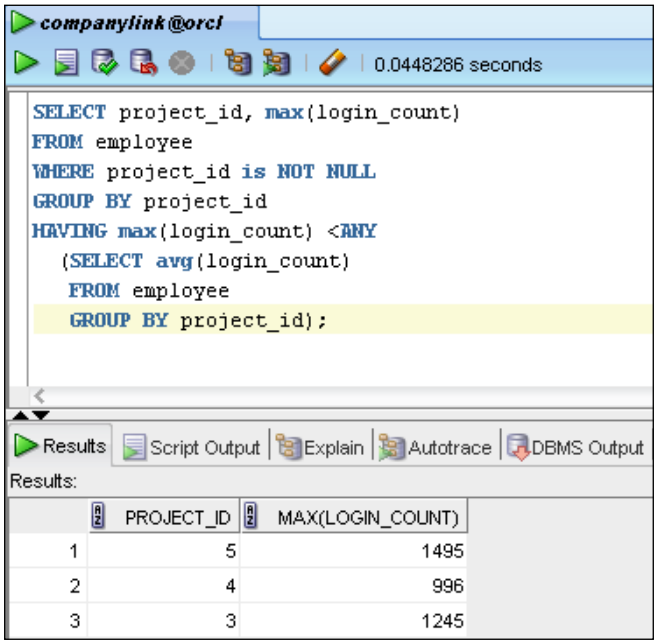
	FIRST_NAME	LAST_NAME	LOGIN_COUNT
1	George	Taylor	798
2	Ken	White	866
3	Helen	Harris	915
4	Kevin	Lewis	996
5	Sandra	Rodriguez	1021
6	Donald	Perez	1025
7	Carol	Clark	1123
8	Matthew	Garcia	1145
9	Daniel	Robinson	1220
10	Laura	Thomas	1221
11	Linda	Anderson	1245
12	Cynthia	Hall	1478
13	Gary	Moore	1495
14	Lisa	Lee	1945
15	James	Johnson	2135

Keeping track of the rules for `ANY` and `ALL` can be difficult. The following table should make this easier:

Type of operator	Type of value returned
<ANY	Values less than the highest value returned by the subquery
<=ANY	Values less than or equal to the highest value returned by the subquery
>ANY	Values greater than the lowest value returned by the subquery
>=ANY	Values greater than or equal to the lowest value returned by the subquery
<ALL	Values less than the lowest value returned by the subquery
<=ALL	Values less than or equal to the lowest value returned by the subquery
>ALL	Values greater than the highest value returned by the subquery
>=ALL	Values greater than or equal to the highest value returned by the subquery

Using multi-row subqueries with HAVING clauses

At times, it is advantageous to use subqueries with `GROUP BY` statements in order to see data at a grouped level. As we have seen, row group exclusion requires the use of the `HAVING` clause. Fortunately, in SQL, the `HAVING` clause can be used with subqueries in ways similar to the `WHERE` clause. We can use multi-row subqueries that return grouped data to an outer query that is also grouped. Those subquery groupings can then be used to restrict the output of the outer query's groups. An example of this is shown in the following screenshot:



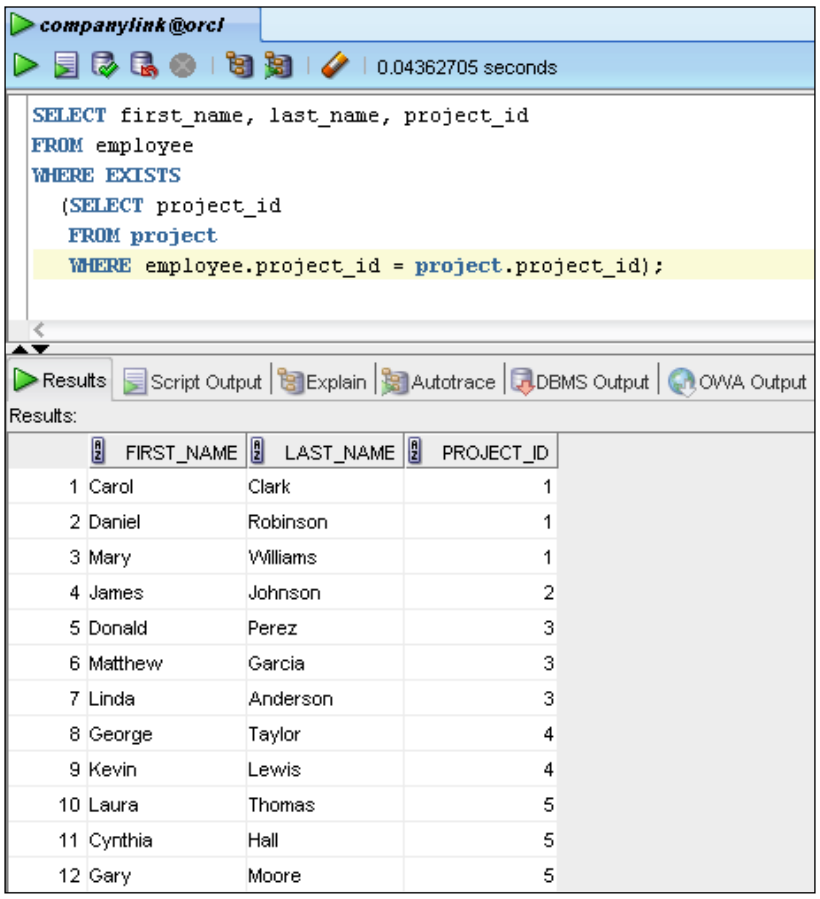
Here, our subquery groups the average `login_count` for all employees by `project_id`. This returns six values to the outer query. The outer query also selects the `project_id` and `login_count`, but it groups them by the maximum value for `login_count` where the `project_id` is not null. The `HAVING` clause restricts the outer query output to only those values that are less than the highest value in the subquery. In simpler terms, the query displays maximum `login_count` values that are greater than the average, grouped by `project_id`.

These subqueries are complex. If you are having difficulty understanding them, you could try running them separately. First, type and run the subquery by itself to see its results, then put it back into the entire statement.

Using correlated subqueries

So far, the subqueries we have used operate under a simple rule. In the process of statement execution, the subquery is executed first, then its results are passed back to the outer query. The technical name for this group of subqueries is a **non-correlated subquery**. We also have another type of subquery at our disposal—the **correlated subquery**. The correlated subquery executes a statement in the reverse pattern of a non-correlated subquery. It executes the outer query first, then for each value returned must evaluate whether the value from the outer query matches one from the subquery. If it does, the row is displayed. If not, it continues to the next value in the outer query and repeats the process. A correlated subquery, then, evaluates the subquery once for every row in the outer query. Thus, if the outer query produces 1,000 rows, the subquery is executed 1,000 times—once for each row.

The correlated subquery is invoked using a technique rather than any particular syntax. A subquery behaves as a correlated subquery when the subquery references a column in the outer query. Because the correlated inner query references an outer column, it cannot be executed to completion first, as with a non-correlated subquery. This behavior forces the statement to evaluate each outer value first. The correlated subquery often makes use of a new operator—`EXISTS`. The structure of the statement using the correlated subquery is very similar to any subquery that is paired with a `WHERE` clause, as shown in the following screenshot:



	FIRST_NAME	LAST_NAME	PROJECT_ID
1	Carol	Clark	1
2	Daniel	Robinson	1
3	Mary	Williams	1
4	James	Johnson	2
5	Donald	Perez	3
6	Matthew	Garcia	3
7	Linda	Anderson	3
8	George	Taylor	4
9	Kevin	Lewis	4
10	Laura	Thomas	5
11	Cynthia	Hall	5
12	Gary	Moore	5

In this example, the subquery appears to join the employee and project tables on `project_id`. However, notice that the subquery's `FROM` clause only references the project table. Thus, the `employee.project_id` column in the join cannot be referencing the `employee` table directly. To see this in action, try to execute the subquery by itself. The statement will produce an error as the join references a column in the `employee` table without listing it in the `FROM` clause. The only conclusion we can draw is that the `employee.project_id` is actually coming from the outer query. This is indeed the case and is what makes the correlated subquery unique. Correlated subqueries can also use operators such as greater than (`>`) and less than (`<`). Remember that it is not the operator used that correlates a subquery. Instead, it is the method of execution.

SQL in the real world

Correlated subqueries can be notoriously inefficient, owing to the fact that they repeatedly execute the inner subquery for every occurrence of a row in the outer query. Often, correlated subqueries can be rewritten as joins for much faster performance. The following query could replace our correlated subquery example with much better performance. Unless your statement requires execution in this manner, consider another path.

The following set of lines is an example of correlated subquery that can be written for faster performance:

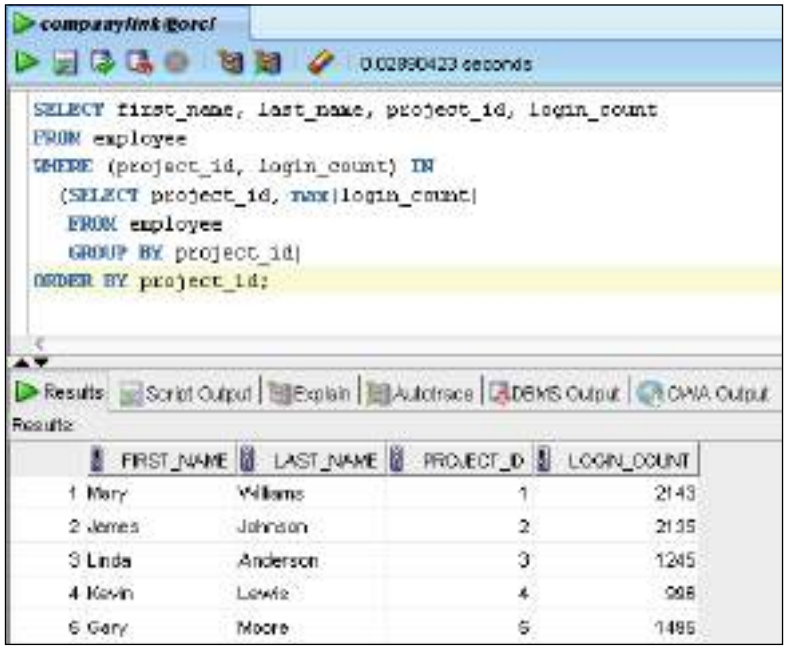
```
SELECT first_name, last_name, project.project_id
  FROM employee, project
  WHERE employee.project_id = project.project_id;
```

Using multi-column subqueries

The subqueries we've seen so far, both single-row and multi-row, have an aspect in common. Whether the values returned are selected from a subquery using `WHERE` or grouped using `GROUP BY` (or any other method), both subqueries only return values from a single column. Taking into account that columns of data represent values that are of a certain type of data, each of these subqueries operate only on a single datatype. However, there is another type of subquery—the **multi-column subquery**—that, albeit more restrictive, allows multiple columns to be processed within a subquery.

Using multi-column subqueries with WHERE clauses

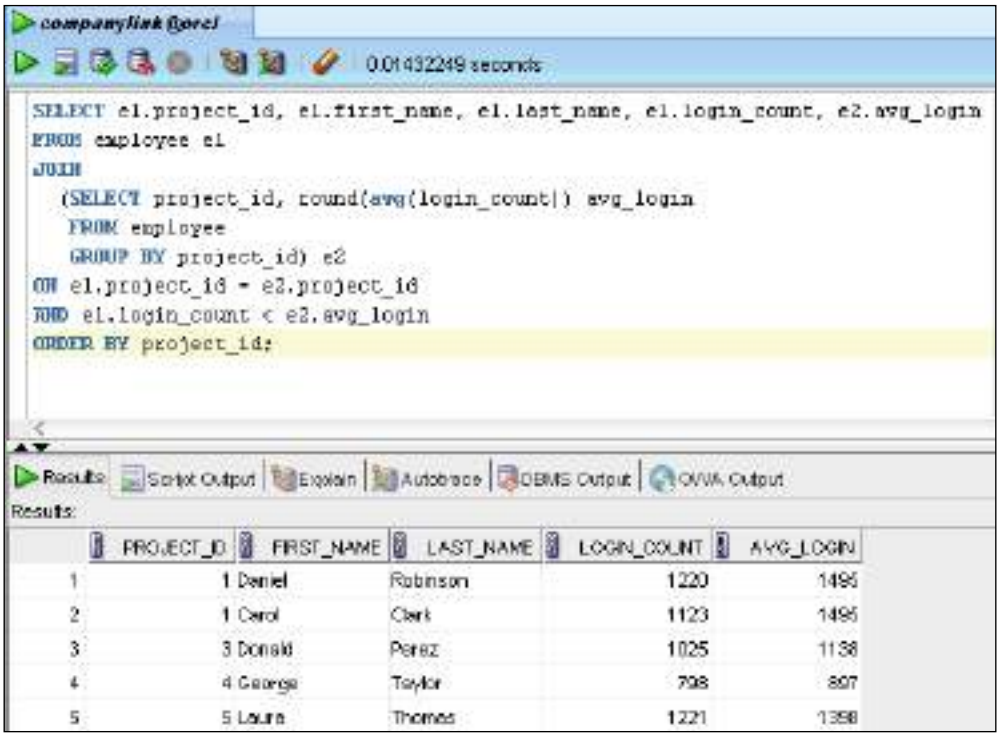
Multi-column subqueries can be used with the `WHERE` clause and the `IN` operator to evaluate multiple column values in the outer query against multiple columns in the subquery. With this method, we can select both the grouped column and the grouping function and return them to the outer query; something we are unable to do with single and multi-row subqueries. An example of this is shown in the following screenshot:



Here, the subquery returns the highest login counts grouped by `project_id`. Then, the outer query matches `first_name`, `last_name`, `project_id`, and `login_count` against that list of values. In realistic language, this query displays the name, project, and login counts of the employees with the highest number of logins for each project. Notice two syntactical points about this statement. First, the order of the columns selected in the subquery is very important. The column `order`, `project_id` first, followed by `login_count`, must match the order of two columns specified in the `WHERE` clause. Failure to construct the subquery in the proper order will result in an error caused by mismatched datatypes or, in our case, no values returned. The second syntactical rule to remember is that, in the `WHERE` clause, the multiple columns specified must be enclosed in parentheses. Failure to do so will result in a syntactical error.

Multi-column subqueries with the FROM clause

The second type of multi-column subquery involves a unique method that can be used to combine subqueries and joins in a single statement. When using multi-column subqueries in the `FROM` clause of a SQL statement, the subquery forms a set of values that is referred to as an **in-line view**. It is often easier to think of this in-line view as a pseudo-table that exists only during the execution time of the statement. It is not a true table, nor is it an actual temporary table. Both of these can exist as database objects. The pseudo-table, however, can be joined with the table referenced in the outer query to produce some unique datasets. In the following screenshot, we show an example of a statement using a multi-column subquery in the `FROM` clause:



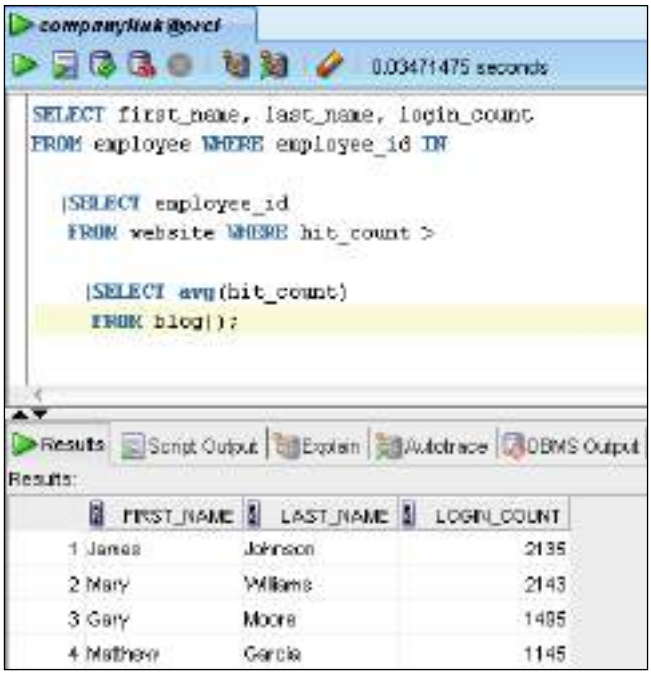
There is a lot going on in this query, so let's take it one piece at a time. Recall your join syntaxes from *Chapter 5, Combining Data from Multiple Tables*. First, let's identify the first table we are joining and its columns. From the `employee` table, aliased as `e1`, we select `project_id`, `first_name`, `last_name`, and `login_count`. Second, let's identify the second table, aliased as `e2`**.** The second table in the join is not a true table at all—instead, it is a pseudo-table, or in-line view, formed by the subquery. That subquery selects two columns—the `project_id` and the rounded average `login_count` grouped by `project_id`. Notice that we alias one of the columns—`round(avg(login_count))`– as `avg_login`. Thus, the `e2.avg_login` column listed in the `SELECT` clause comes from the subquery. The results of this pseudo-table can be joined to the data from the `employee` table just as with any other join. In our case, we use the `JOIN...ON` syntax to do this. Having joined the data by `project_id`, we then restrict the results using a `WHERE` clause, allowing only row values with a `login_count` less than the average to be displayed. We complete the statement by ordering the output based on `project_id`. In natural language, this statement could be phrased as, *Display project, name, and login count information for employees that have a login count that is lower than average for their project*.

Investigating further rules for subqueries

Aside from the basic differences involved with using subqueries, there are a few additional facts we should remember. We take a look at these rules in this section.

Nesting subqueries

A subquery is a query that is nested within another query. However, subqueries can also be nested inside other subqueries. Multiple **nested subqueries** allow us to use queries from more than two tables, although they can also be used with a single table. An example of this is shown in the following screenshot with indentation and line separation to make the different queries more distinct:



Here, we have three distinct queries—one from the blog table, one from the website table, and one from, employee. The innermost query selects the average of the `hit_count` column from blog and passes it back to the second level query. That query returns `employee_id` values that have a `hit_count` in website that are greater than the average `hit_count` in blog. The outermost query displays `name` and `login_count` information for employees that meet the criteria of the second level query. A query such as this might be used to see if there is any correlation between an employee's logins and hits to their websites and blogs.

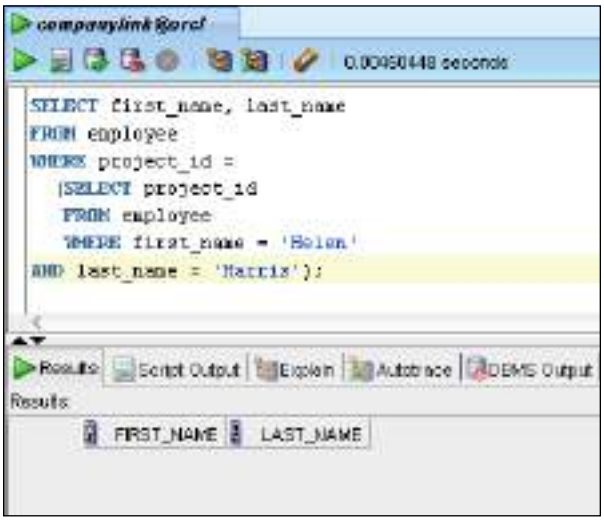
If we can nest multiple subqueries within each other, how far can we go? For all practical purposes, there is almost no limit. Strictly speaking, we can nest subqueries in `WHERE` clauses to a depth of 255. There is no limit to how many we can nest in `FROM` clauses.

SQL in the real world

In looking at our nested subquery example, you may think, *How could I ever write something like that?* Although nested subqueries can be complex, remember that they are just `SELECT` statements inside of each other. Begin by writing the innermost query first and work outward. Run the subqueries separately and look at the results. Always pay particular attention to indentation and your use of parentheses. These tips can make writing complex subqueries much easier.

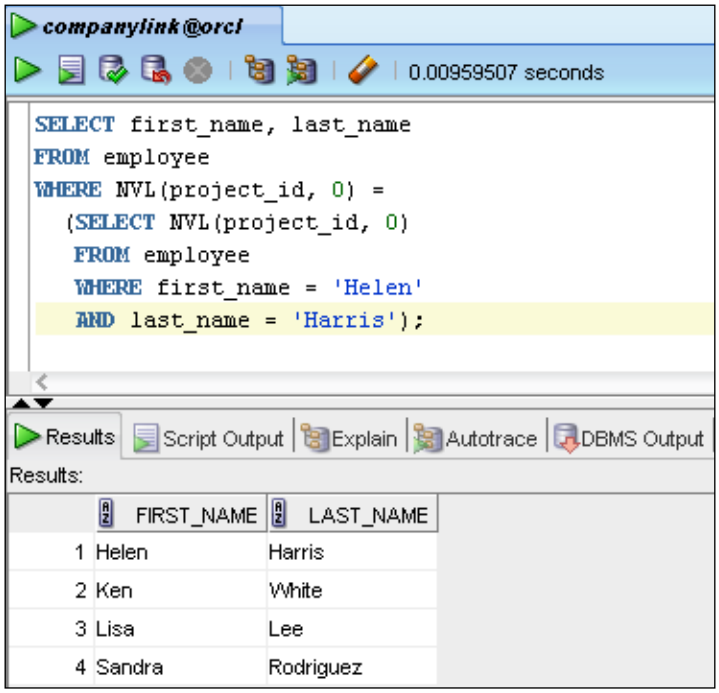
Using subqueries with NULL values

One of the most common traps that new SQL programmers fall into when writing subqueries is the problem of nulls. As we've mentioned before, a `NULL` is not an actual value. Rather, it is the lack of a value. When a subquery returns a `NULL` to the outer query, it cannot be evaluated and, thus, returns no rows. We see an example of this in the following screenshot:



There is certainly a more direct way to write this query. Its purpose is to display the first and last name for employees that have the same `project_id` as `Helen Harris`. However, it is a good example of how information can be lost in a subquery. As `Helen Harris` has a `NULL` value for `project_id`, no data is returned to the outer query and, consequently, no rows are returned from the statement.

We could, however, rewrite this statement using a single-row function to deal with the nulls. Recall that the function `NVL()` can be used to substitute a value for a `NULL`. If we place an `NVL()` function in both the subquery and the `WHERE` clause of the outer query, we can get the desired data. In the next example, we substitute a zero for the nulls in order to properly evaluate the statement:



SQL in the real world

As our SQL statements grow more complex, it's easy to lose sight of what we are actually doing. In actuality, we are using fewer literal values and instead using statements to extract conditions for us. These conditions have a greater flexibility than literals, allowing for different conditions each time the statement is run. SQL statements such as these encapsulate an idea instead of just code. A true SQL expert learns how to bridge the gap between business logic and SQL commands.

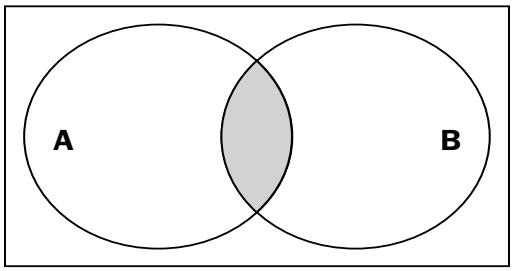
Using set operators within SQL

Our last method of combining queries is through the use of set operators. While set operators are generally not used by SQL programmers on a day-to-day basis, when the situation calls for them, they can be extremely useful. Before we examine SQL set operators in detail, it is advantageous for us to briefly review mathematical set theory to see how they work.

Principles of set theory

If you think back to your school math classes, you may remember learning about a topic known as set theory. **Set theory** is the branch of mathematics that involves the study of collections of objects, or sets. Because databases deal with collections of data, set theory is relevant to the more theoretical aspects of relational database theory. However, for our purposes, an extensive review is not necessary. It is enough to learn the basic ideas and terminology of set theory as it applies to SQL.

Set theory is used to describe the interaction of groups of objects or lack thereof. The primary method used to model this behavior graphically is the **Venn diagram**. A Venn diagram describes various sets of objects and their relationships. An example of a Venn diagram is shown as follows:

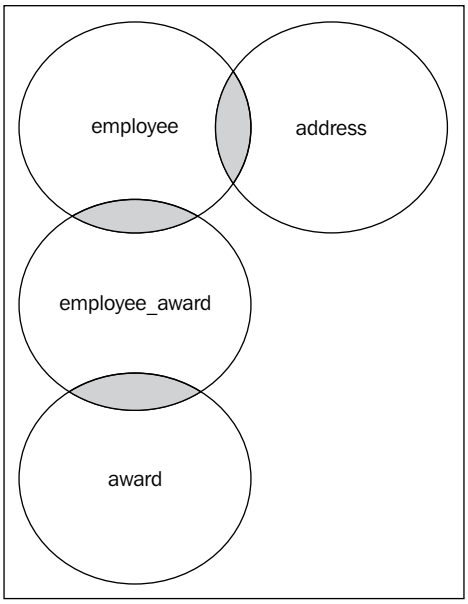


Here, we have a diagram representing two sets of data, denoted with A and B. We refer to these as set A and set B. The diagram indicates that A and B overlap, shown with the shaded area. This overlap indicates that A and B share common members between the two groups and is referred to in set theory as the **intersect** or **intersection** of A and B.

Set theory uses many terms to represent these interactions, but only a few are relevant to our purposes. Besides intersection, the other primary term we use to describe these relationships is union. **Union** is defined as a combination of all members in both set A and set B. Therefore, intersection is exclusive, while union is inclusive.

Comparing set theory and relational theory

Set theory has a special relevance to our study of SQL in relational databases. This is because we can express tables as sets of data and the relationships between them in the terminology of set theory. In the following diagram, we modify our Venn diagram to relate to the data structures of our `Companylink` database:



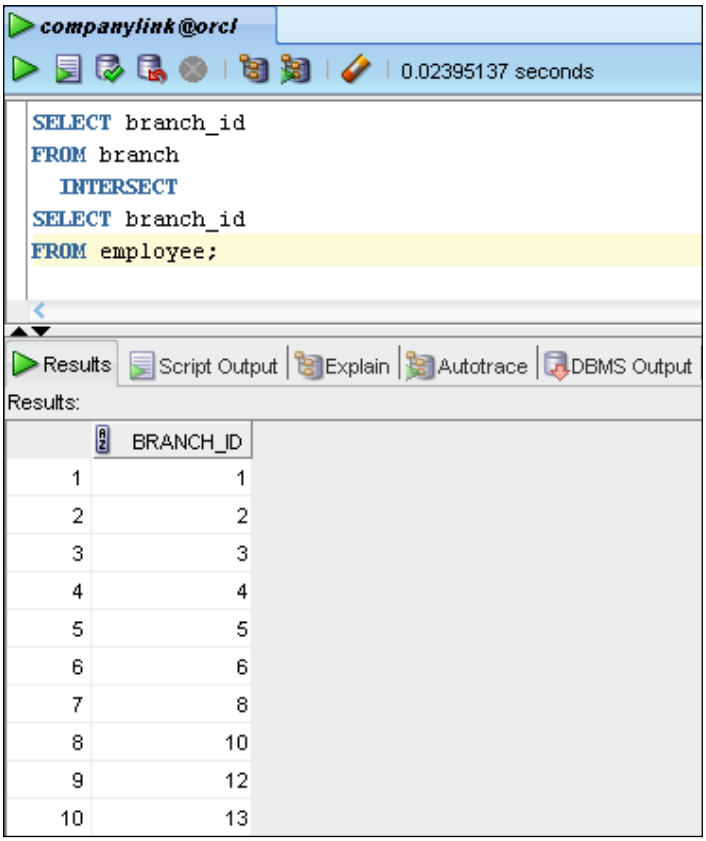
In our diagram, we display the interactions between the employee, address, `employee_award`, and `award` tables. Notice that some tables, such as address and award, have no interaction, while others, such as address and employee, show an overlap. The overlap indicates common members between the two sets; in the case of employee and address, both share similar values in the `employee_id` column. We could also, of course, include all the `Companylink` tables in a model such as this to show the full range of their interaction, but generally an ERD is more suited to such a purpose. It is crucial to understand that, while an ERD displays relationships between table columns, set operations deal with rows of data. Also note that a set need not be an entire table. In fact, in SQL, we use `SELECT` statements to represent these sets.

Understanding set operators in SQL

In SQL, principles of set theory are applied using set operators. **Set operators** are SQL operators that allow data to be expressed in terms of the primary operations of set theory. There are four principle set operators that we can use to combine the data from `SELECT` statements. Each statement that uses a set operator will include at least one set operator and at least two `SELECT` statements. We look at each of the operators here.

Using the INTERSECT set operator

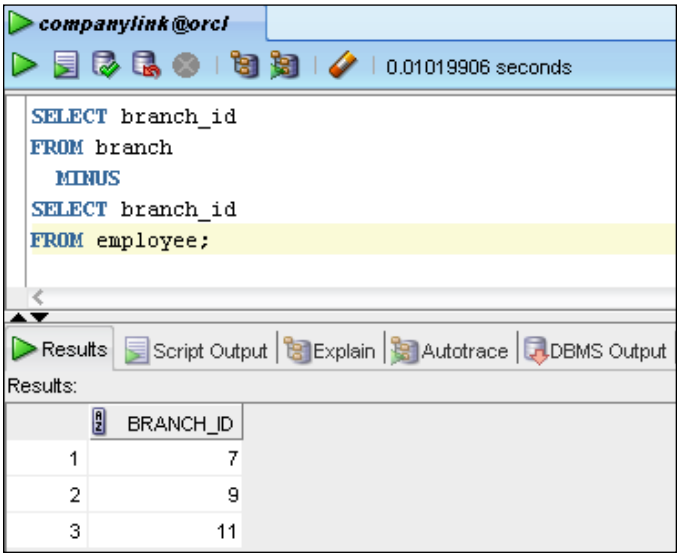
An `INTERSECT` operator returns all rows that are common to both `SELECT` statements. In set theory, it is the intersection between two datasets. An example of `INTERSECT` is shown in the following screenshot:



This statement executes each `SELECT` individually and then combines the results using `INTERSECT`. The first `SELECT` returns 13 rows—the total number of `branch_id` values in the branch table. The second `SELECT` statement returns 16 rows as each employee is assigned to a branch using a value for `branch_id`. When we find the intersection between them, 10 rows are returned—the set of common `branch_id` values between employee and branch. This indicates that there are values for `branch_id` in the branch table that are not used in the `employee` table, as the intersection returns fewer rows than the total number in the `branch` table. In short, there are 13 possible `branch_id` values in the `branch` table, but only 10 are used in the `employee` table. We can re-state this query in more common language as, *Display a discrete list of the branch IDs for all employees*.

Using the MINUS set operator

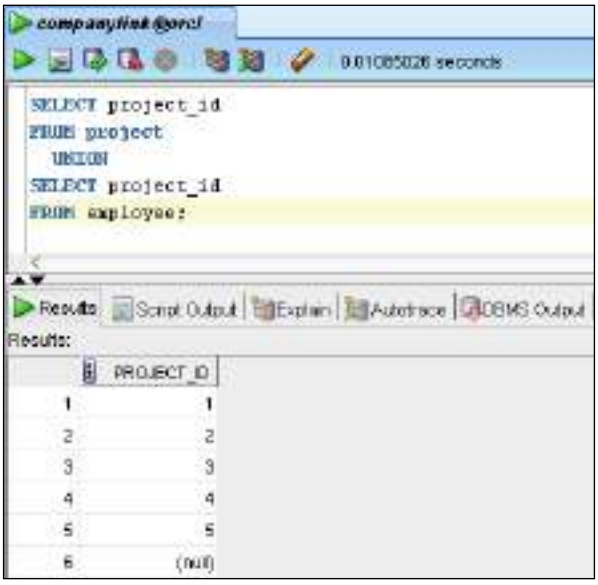
When we need to subtract one set of values from another, we can use the `MINUS` operator. `MINUS` removes all values from the second `SELECT` statement that are also found in the first `SELECT`. Say that we are asked for a list of `branch_id` values that have not been assigned to any of the employees in the `employee` table. As the following screenshot shows, we can use `MINUS` to perform the given query:



Here, we have rewritten the previous query to subtract `branch_id` values in the `employee` table from the `branch` table. The first statement retrieves all possible values for `branch_id` from the branch table. The second statement selects all the values that are assigned to employees from the `employee` table. When `MINUS` is used to remove all the assigned values from the possible values, only the unassigned ones remain. Note that the order of statements is very important when using `MINUS`. Were we to change the order of the two `SELECT`s in this query, no rows would be selected. Such a statement would remove all possible values from assigned ones, leaving no values.

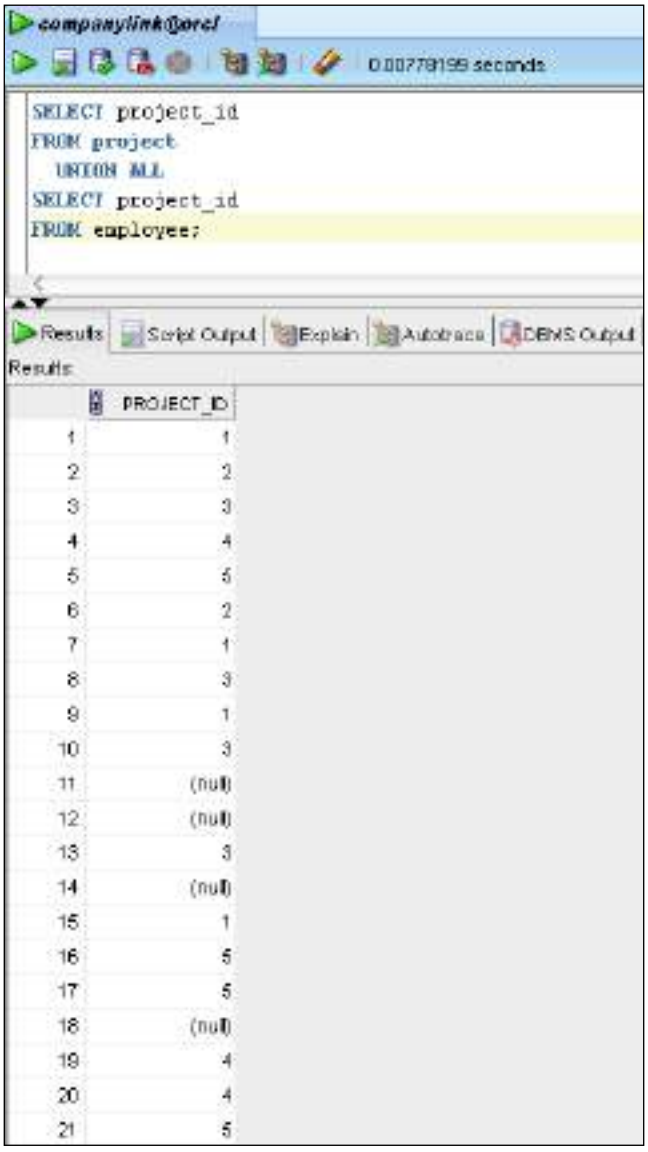
Using the UNION and UNION ALL set operators

Our last examples of set operators are perhaps the most commonly used. The `UNION` and `UNION ALL` operators are used to combine the result sets from two queries. They do so, however, in different ways. First, let's look at an example using `UNION`, which is shown as follows:



In this statement, the first query selects all values for `project_id` from the `project` table. The second does the same for the `employee` table. The `UNION` operator combines the two result sets and displays them. However, in doing so, it removes duplicate values. Thus, the results include all possible values for `project_id` from both the `project` and `employee` tables. Note that this result set also includes one `NULL` value, as the `employee` table contains `project_id` values that are null.

The `UNION ALL` operator performs a very similar function, with one major difference. Whereas the `UNION` operator suppresses duplicate values from the results, `UNION ALL` displays all values including duplicates. In our next example, we rewrite the previous query with one simple difference—we change the `UNION` operator to `UNION ALL`. The results are quite different:



SQL in the real world

In terms of SQL performance, the `UNION ALL` operator is almost universally preferred to `UNION`. Given the large number of values resulting from a `UNION ALL`, this may seem strange. It is, however, true. As we've noted, a `UNION` performs the added step of removing duplicate values. As it turns out, this is a very costly operation in terms of performance. Where possible, you should generally use `UNION ALL` instead of `UNION`.

Summary

In this chapter, we have learned two new ways to combine sets of data. First, using the subquery, we learned to nest queries inside of each other to combine data from tables without a direct relationship. We looked at ways to manipulate scalar subqueries that return a single value, multi-row subqueries that return multiple values, and multi-column subqueries that return values from more than one column. We followed this up with a look at using SQL set operators to combine overlapping and non-overlapping data.

In this chapter, we've come to the end of our exploration of the use of `DML` statements and techniques. However, although we have learned countless ways to manipulate table data, we have yet to learn how to actually create a table from scratch using SQL. The next chapter will introduce the concept of database object creation using a sublanguage of SQL known as **DDL** (**Data Definition Language**).

Certification objectives covered

In this section, we have seen the following certification objectives covered:

- Define subqueries
- Describe the types of problems that the subqueries can solve
- List the types of subqueries
- Write single-row and multiple-row subqueries

Test your knowledge

1. Which of the following is NOT a type of subquery?
   a. Single-row
   b. Multi-row
   c. Multi-column
   d. Multi-scalar
2. What is the output of the following query?

```
SELECT blog_url, min(hit_count)

FROM blog;
```

   a. 123
   b. 18
   c. 0
   d. A "not a single-group function" error is returned.
3. What is the outcome of the following statement?

```
SELECT blog_url, hit_count

FROM blog

WHERE hit_count =

  (SELECT blog_url, max(hit_count)

  FROM blog

  GROUP BY blog_url);
```

   a. An error is returned because too many columns are selected in the outer query.
   b. An error is returned because too many columns are selected in the inner query
   c. The statement returns multiple values
   d. The statement returns a single value

4. Which of the following can be used to create a derived column?
   a. A scalar subquery with the WHERE clause
   b. A scalar subquery with the HAVING clause
   c. A scalar subquery with the SELECT clause
   d. A multi-row subquery using the IN operator
5. Given a multi-row subquery is used in a statement, which operator will match values in the outer query that are less than the highest value returned by the subquery?
   a. <ANY
   b. >ANY
   c. <ALL
   d. >ALL
6. In a non-correlated subquery, which subquery is evaluated first?
   a. The inner query
   b. The outer query
   c. Neither query
   d. Both queries are evaluated simultaneously
7. In a correlated subquery, which subquery is evaluated first?
   a. The inner query
   b. The outer query
   c. Neither query
   d. Both queries are evaluated simultaneously
8. A SQL programmer intends to make use of a subquery that forms an in-line view. In order to do this, the subquery must be used with what clause?
   a. FROM
   b. WHERE
   c. HAVING
   d. ORDER BY

9. What is the maximum number of nested subqueries that can be present in a WHERE clause?
   a. 2
   b. 3
   c. 128
   d. 255

10. What function is often used to mitigate the negative effects of NULL values in a subquery?
    a. AVG()
    b. TO_CHAR()
    c. NVL()
    d. Any multi-row function

11. Which SQL set operator can be used to subtract the results of one query from another?
    a. INTERSECT
    b. MINUS
    c. UNION
    d. UNION ALL

12. Which SQL set operator can be used to combine two result sets and while removing duplicates?
    a. INTERSECT
    b. MINUS
    c. UNION
    d. UNION ALL

13. Which SQL set operator can be used to return all values that are common to two SELECT statements?
    a. INTERSECT
    b. MINUS
    c. UNION
    d. UNION ALL

14. Which SQL set operator can be used to combine two result sets and without removing duplicates?
    a. INTERSECT
    b. MINUS
    c. UNION
    d. UNION ALL

9
Creating Tables

Throughout this book, we've seen numerous ways to manipulate table data through the use of DML. But, up to this point, we've only manipulated existing tables. It's time to explore object creation with DDL—Data Definition Language—and how it can be used to create database tables that support business rules.

In this chapter, we shall:

- Discuss Data Definition Language (DDL) and its purpose
- Understand Oracle's schema-based approach
- Examine the structure of tables and datatypes
- Describe the `CREATE TABLE` syntax
- Use `CREATE TABLE AS..SELECT` to copy tables
- Examine the purpose of constraints
- Understand and use different types of constraints

Introducing Data Definition Language

In *Chapter 4, Data Manipulation with DML* we established a distinction between different types of sub-languages in SQL. We pointed out that DML, or Data Manipulation Language, is used for operations such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. We also mentioned that another sub-language, DDL, can be used for other purposes.

Understanding the purpose of DDL

DDL, or **Data Definition Language**, is used primarily to create objects in the database. A **database object** is any persistent entity within the database that forms a part of its logical structure. The most common object within a database is the table. We refer to a table as a logical object since it has no actual physical structure, but once it is created, it exists until it is removed. This persistence allows database objects created using DDL to be used repeatedly, even after a database is shutdown and restarted.

In terms of syntax, the scope of DDL is significantly larger than that of DML. DML is encapsulated, for the most part, in four basic clauses—`SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Those four clauses are then used to manipulate data. A DDL statement, on the other hand, can use hundreds of possible clauses. Fortunately for us, many of these clauses are extremely uncommon and all of them fall into three basic operations—`CREATE`, `ALTER`, and `DROP`. Each of these is paired with a database object to form clauses such as `CREATE TABLE` or `ALTER INDEX`.

Examining Oracle's schema-based approach

Before we look at the syntactical structure of DDL, it is important that we understand how Oracle organizes the ownership of database objects. The ownership of objects is crucial to how security is managed in a database. While an extensive look at security is beyond the scope of this book, we will examine the basics of object ownership in order to better understand how tables are created.

Some database systems make a distinction between database users and object owners. In these systems, a **database user** is defined as an account that allows access to the database. These types of accounts are common today. We use them any time we log in to a system, such as an e-mail account or web portal. Converse to this is an **object owner,** which, in a system that makes such a distinction, is an account that exists only to create and own database objects. For instance, a system might have a user called `finance`. The finance user does not represent any particular person who logs in to the system; rather, we use the finance user when we want to create tables that hold financial data. We then say that those tables are *owned* by the finance user.

In Oracle, no such distinction between users and owners exists. One type of account exists that can be used for both purposes. For instance, continuing with the example of financial tables, we could create a *finance* user. This is a true database user, which can be given the proper privileges to access and manipulate the system. However, we can also use this user to log in and create database objects, such as financial tables. Any database objects that are created while we are logged in as the finance

user are said to be *owned* by that user. When a database user owns objects, it is referred to as a **schema**. A schema is the user account and the collection of all the objects that it owns. Thus, if we log into the database as the finance user and create a few tables, we can say that those tables are owned by finance and form the "finance schema".

In *Chapter 1*, *SQL and Relational Databases* we created a connection to the database in SQL Developer that uses the `companylink` user to create a session to the database. The script you ran to create the database tables also runs under this user. Thus, the tables we have used in our examples have all been within the `companylink` schema.

SQL in the real world

Some have argued that Oracle's dual user/owner approach is inherently less secure than separating the two. Even if this is true, it only means that extra care must be taken when administering users. In Oracle, accounts can be locked. This prevents them from being used to access the database, but still allows them to own objects. Some DBAs design systems that force a separation of the two, allowing only locked accounts to own objects. The Oracle model allows for flexibility, but requires a greater understanding of security administration.

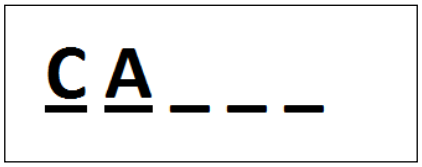
Understanding the structure of tables and datatypes

In earlier chapters, we discussed the way that tables are constructed; namely, with rows and columns. To review, a row contains a single *occurrence* of information. For instance, one row from the `employee` table provides us with all the information for a single employee, including name, gender, date of birth, and other information. A column, on the other hand, represents one *type* of information. For example, the `message_textcolumn` in the message table contains the textual information for each message sent, regardless of which employee sent it. The `message_text` column contains no other data. It does not contain the employee ID of who sent it or the date it was sent. That information is contained in other columns. Thus, we can say that a column will always contain a certain type of data and can contain no other. The `message_text` column contains character string data. As such, it cannot contain other information types such as dates or numeric values. If other values such as numbers are contained within the text of the message (which is certainly possible), they are treated as character values. They cannot be used, for instance, in numeric functions, without first extracting them from the other character values. We can say that the data in each column is constrained by its datatype. A **datatype** is a way of classifying the types of data that are possible in a particular column. The datatype of a column is defined as a part of the creation process. A column cannot be created

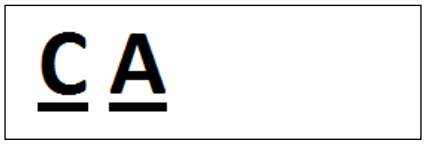
without a datatype. When we define the datatype for a column, we also define its scope, such as the maximum length for a character string in the column. While the type of scope is different depending on the datatype, it is usually enclosed in parentheses. There are many datatypes available to us in Oracle, but there are five that are most commonly used.

CHAR

Data defined with the **CHAR** datatype is a fixed-length alphanumeric data. By fixed-length, we mean that data of type `CHAR` is non-varying in length. For instance, say that we define a "State" column in a table called `address`. The State column is used to hold the two-digit State abbreviation for each address. We could define this column as being of type `CHAR(5)`—a fixed-length column holding alphanumeric data that is five characters in length. We could display the way that a piece of information is stored in the table as shown in the following figure:



Here, we see that the abbreviation for California, CA, is stored in the first two positions. Since **CA** only requires two characters, the remaining three positions are padded with spaces, owing to the fact that `CHAR` is fixed-length. A `CHAR(5)` will always contain five characters, whether they are used or not. In our example, the extra spaces are essentially useless, since our State codes will always be two characters in length. Thus, if we decide to use the `CHAR` datatype for our State codes, it would be better to use a `CHAR(2)`. This prevents the storage of unnecessary spaces. A visual representation of the way it would be stored is shown as follows:



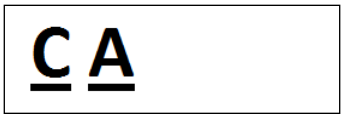
The `CHAR` datatype has a maximum length of 2,000 characters.

SQL in the real world

While Oracle continues to support the `CHAR` datatype, it is generally recommended that you avoid using it. There have been reports of problems with certain tools and features involving `CHAR`. While you should still be aware of it because of its continued use, you should generally use our next datatype, `VARCHAR2`, for character data.

VARCHAR2

Data defined with the **VARCHAR2** datatype is alphanumeric data of a variable length. Unlike the fixed-length `CHAR` datatype, `VARCHAR2` only stores the number of characters that are contained in any particular value. Thus, a `VARCHAR2(5)` is variable-length character data that is *up to* five characters in length. Returning to our previous example, let's say that we decided to define our State column as a `VARCHAR2(5)` instead of a `CHAR(5)`. We can visually represent this as shown here:



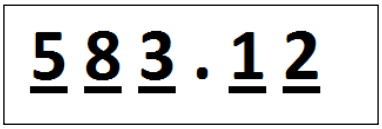
As we can see, even though our `VARCHAR2` is defined to hold up to five characters, only two are required. Since `VARCHAR2` is a variable-length datatype, it *chooses* to only allocate two positions of data for the two characters required. If for some reason we needed to update this value to one that requires more than two characters, additional positions will be allocated, provided that the number of positions required is not greater than its defined limit—in this case, five characters. This makes `VARCHAR2` a much more flexible datatype for character data than `CHAR`. It also has a larger maximum value. The `VARCHAR2` datatype can hold up to 4,000 characters.

SQL in the real world

If you're wondering why `VARCHAR2` was named as it is instead of just `VARCHAR`, it is because a datatype called `VARCHAR` already existed. Both were introduced around version 6 of the Oracle database, although the history of `VARCHAR` is somewhat shrouded in mystery. It was likely originally added to conform to the ANSI SQL standard, but `VARCHAR2` gained prominence because of its flexibility. `VARCHAR` has possessed different characteristics through different versions, and you can still use it. However, in Oracle 11*g*, if you define a column as type `VARCHAR`, it will be automatically converted to `VARCHAR2`. There's some indication that Oracle is reserving `VARCHAR` for future use, so it's definitely a datatype to stay away from for now.

NUMBER

NUMBER is Oracle's primary datatype for storing numeric data. It is defined using the format `NUMBER(p,s)`, where p stands for *precision* and s represents *scale*. Precision is defined as the total number of digits allowed for storage of the number, while scale is the number of digits allowed *after* the decimal point. Using these two attributes, NUMBER can be used to specify limits for both integers and real numbers, rather than requiring two separate datatypes. For example, let's say we want to define a column to hold pricing data. This type of data will require that we store real numbers—numbers in decimal format. To do this, we could designate a column, `price`, and define it as a `NUMBER(5,2)`. If this column held the value 583.12, we could represent it as shown in this examp.le.



As we can see, the value 583.12 fits into a `NUMBER(5,2)` column since there are a total of five digits allowed with two of those digits to the right of the decimal point. Note that the precision and scale are the *maximum* values that can be used in the column. Thus, values such as 34.1, 8.54, and even 40 are allowable. Having a scale defined for a number does not require that the value has a decimal component. It simply constrains it as a maximum allowed value. A `NUMBER(5,2)` also implies that there are only three digits to the *left* of the decimal point. Thus, numbers such as 2396.1 are *not* allowed, even though the number has a total of five digits. Any attempt to insert any values that are outside the bounds defined by the precision will result in an error. If a value is within the specified precision, but outside the scale, such as 43.234, the value is automatically rounded to fit within the scale.

Integers can also be defined by omitting the scale for the datatype. We can define a column as `NUMBER(8,0)` or `NUMBER(8)`. When the scale is zero or omitted, the values allowed can contain up to a total of eight digits. Thus, values such as 4093, 920490, and 20940958 are allowed. In the case of a `NUMBER(8)`, any values entered as decimals will be rounded to the nearest integer. The `NUMBER` datatype can also be defined without any precision or scale as simply `NUMBER`. When `NUMBER` is used without precision or scale, it defaults to a `NUMBER(38)`—an integer with up to 38 total digits.

SQL in the real world

Although not often used, the scale specified for the `NUMBER` datatype can actually be a negative number, such as -2. When a negative is used for scale, the negative number specifies the number of significant digits to the *left* of the decimal point. Thus, if we define a `NUMBER(5,-2)` and insert the value 234.34, the number is actually stored as the value 200.

The `NUMBER` datatype allows for values between 1 and 38 for precision and values between -84 and 127 for scale.

DATE

In *Chapter 6*, *Row Level Data Transformation*, we made reference to the unique way that Oracle stores date values. We indicated that in Oracle, a date is neither a character string value nor a numeric value. Rather, we said that the **DATE** datatype stores seven bytes of data that contain the amount of time from January 1, 4712 BC. It uses these seven bytes to store century, year, month, day, hour, minute, and second information. Thus, the `DATE` datatype in Oracle stores both date and time information. `DATE` requires no additional scope to specify a maximum value for the data, although the maximum date value that can be stored is December 31, 9999. As we mentioned in *Chapter 6*, *Row Level Data Transformation*, Oracle displays information of type `DATE` using a default format of DD-MON-YY, although we can extract and manipulate the format displayed using functions such as `TO_CHAR()`.

Other datatypes

Oracle allows the use of many other datatypes. Most of these are highly specialized and less commonly used. While we do not discuss them in detail here, the following reference chart displays some of their names and uses:

Category	Datatype	Description
Numeric	FLOAT	Similar to NUMBER – used to floating-point numbers
	INTEGER	Similar to NUMBER with a scale of zero
Date/time	TIMESTAMP	Dates with fractional seconds
	TIMESTAMP WITH TIMEZONE	Dates with fractional seconds and timezone information
	TIMESTAMP WITH LOCAL TIMEZONE	Dates with fractional seconds with time adjusted to the local timezone of the database
	INTERVAL YEAR TO MONTH	Time interval stored in years and months
	INTERVAL DAY TO SECOND	Time interval stored in days and seconds
Large Object	CLOB	Character data storage up to 4 GB
	BLOB	Unstructured binary data storage up to 4 GB
	BFILE	Pointer to an externally stored file

Using the CREATE TABLE Statement

When we want to create a table within the database, we use the `CREATE TABLE` statement. The `CREATE TABLE` statement has evolved through various Oracle versions from fairly simple to staggeringly complex. The complexity allows for many options to be used when building tables. Fortunately for us, the syntax to create basic tables is relatively simple. The basic syntax tree is shown as follows:

```
CREATE TABLE tablename (
column1,  datatype,
column2,  datatype,
columnx,  datatype
);
```

We will expand on this syntax as we learn more about tables, but this is a sufficient place to start. The statement begins with the `CREATE TABLE` clause followed by a table name of our choosing, then an opening parenthesis. Next, we outline the table's column structure specifying a name and datatype for each one, followed by a column and closing semicolon.

Understanding the rules of table and column naming

Although the creator of a table and its columns determines its names, there are certain rules and limits that we must abide by:

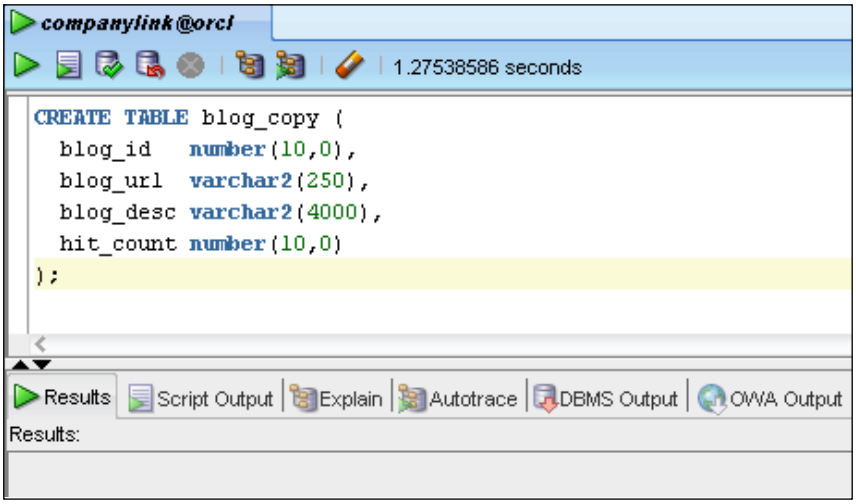
- Names may be between 1 and 30 characters long
- Names may only include alphanumeric characters and the underscore (_), hash symbol (#), or dollar sign ($) characters
- Names must begin with an alphabetic letter between A and Z
- Certain reserved words, such as a name that uses a SQL clause, cannot be used
- Up to 1,000 columns can exist in a table

SQL in the real world

The choice of table and column names is an important one that may depend on your organization's coding standards. Even if you aren't constrained by any particular rules, you should take great care to name your tables and columns in a way that makes them relevant to the data they hold. Such choices as whether or not to allow plurals as names must also be considered.

Creating tables

We have examined the basic syntax for `CREATE TABLE`, so let's proceed with some table examples to see how they work. In the following example, we begin by creating a copy of a table with which we are familiar—the blog table.



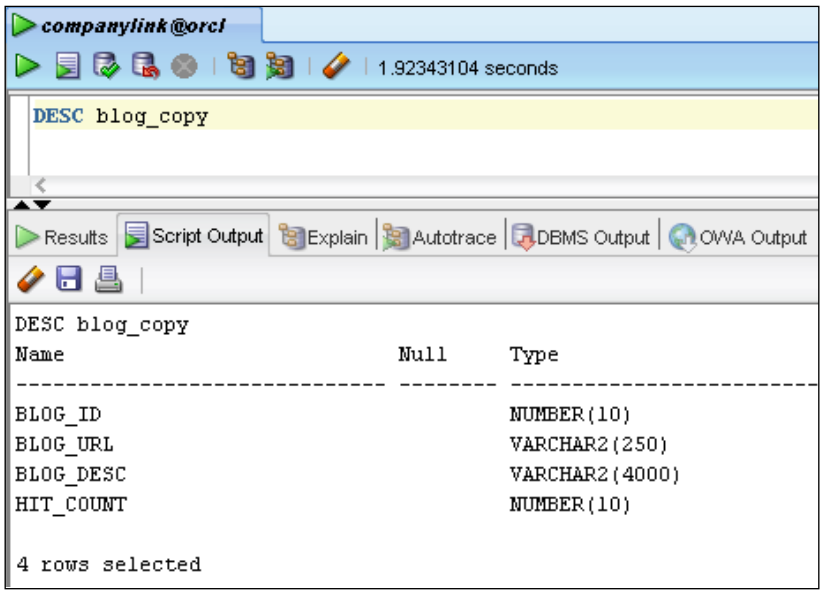
We see that the syntactical form of the previous statement follows that of the syntax tree. It begins with a `CREATE TABLE` clause and is followed by column definitions enclosed in parentheses. For the first column, `blog_id`, we need to choose a datatype that fits the type of data to be stored. Our IDs are numeric integers with no decimals, so we will choose the `NUMBER` datatype with a scope of zero. Our choice for datatype precision depends on how large we anticipate our ID numbers may be. With Companylink, we want to think big, so we'll choose a precision of 10, which allows numbers as large as 9,999,999,999 to be stored. For the `blog_url` column, which can contain a mixture of alphanumeric characters and symbols, we choose the `VARCHAR2` datatype with a maximum size of 250 characters. This should be sufficient to store base domain URLs. The third column, `blog_desc`, will be used to store a full description of the user's blog, so we will make it a `VARCHAR2` as well, with a large maximum of 4,000 characters. Finally, our `hit_count` column contains the number of hits for each blog, so we will choose a non-decimal numeric datatype, `NUMBER(10,0)`.

From the process we've undergone in the previous paragraph, it should be clear that we can only properly create a table when we know fairly specific characteristics about the data it holds, such as the type of data, its maximum possible size, and the number of values possible within the table. At some level, we must undergo this process in order to design an efficient table.

SQL in the real world

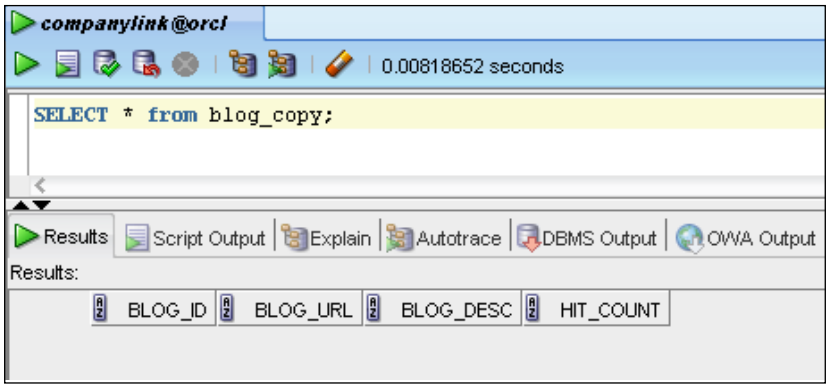
Certain graphical third-party tools exist that can make the process of designing tables faster. Such tools can even generate the SQL needed to create the tables and columns without typing a single `CREATE TABLE` command. However, the thought process involved is the same regardless of the preferred tool. You must understand how the table will be used.

When we run the `CREATE TABLE` statement, we receive a prompt at the bottom that informs us that our statement was successful. If we type the statement incorrectly, we receive an error message. Once our table is successfully created, we can view it in two ways. First, we can use the `DESCRIBE` command, abbreviated as `DESC`.

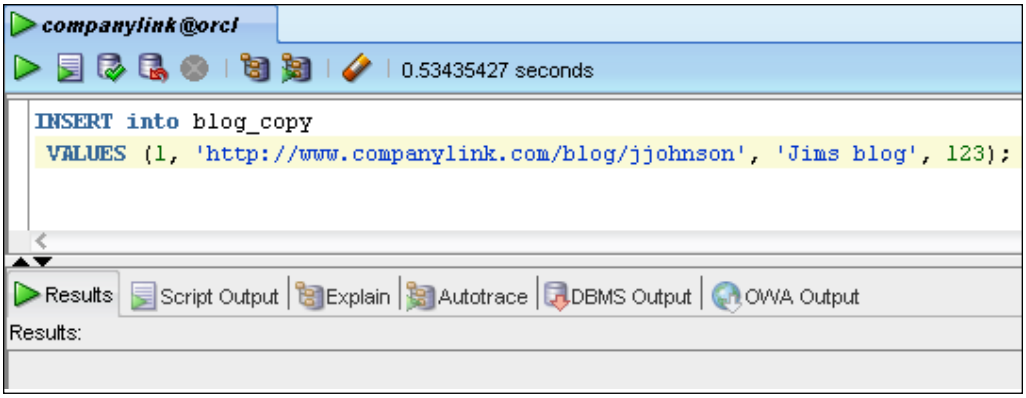


We first saw the `DESCRIBE` command in *Chapter 2*, *SQL SELECT Statements*. It tells us about the structure of the table. We see that each column is listed in the order that we specified, as well as its datatype and scope. In SQL Developer, we also see a column in the description called simply `Null`. We will cover this column later in the chapter.

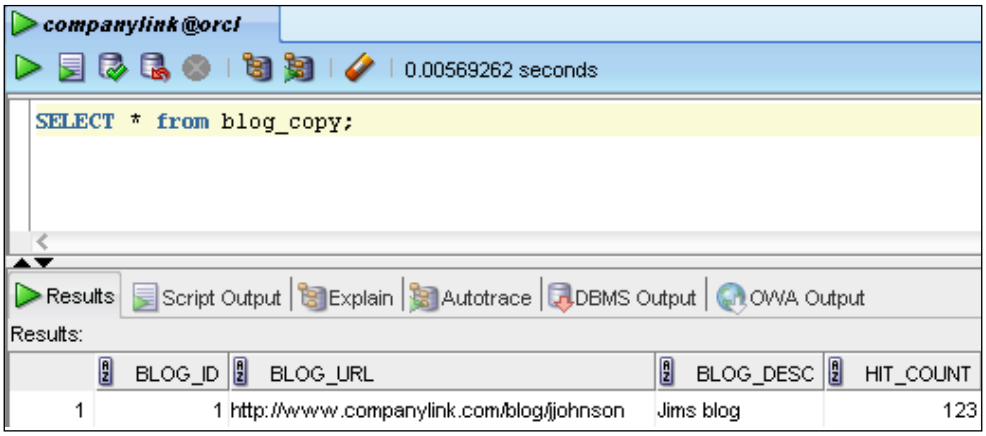
While the `DESCRIBE` command tells us about the structure of a table, notice that it does not display any actual data. As we've seen many times throughout this book, in order to view data, we need to use a `SELECT` statement, as shown in the following example:



As we can see, the statement executes successfully, but no rows are returned. We see only the table column headings. This is the expected outcome, since we haven't yet added any data to our new table. To do that, we need to use the `INSERT` statement that we covered in *Chapter 4*, *Data Manipulation with DML* as shown in the following screenshot:



For the sake of simplicity at this point, in the next example we are adding the data from the first row of the blog table to our `blog_copy` table. Once we successfully insert the row, we can view the structure and data of our first manually created table using a `SELECT` statement.

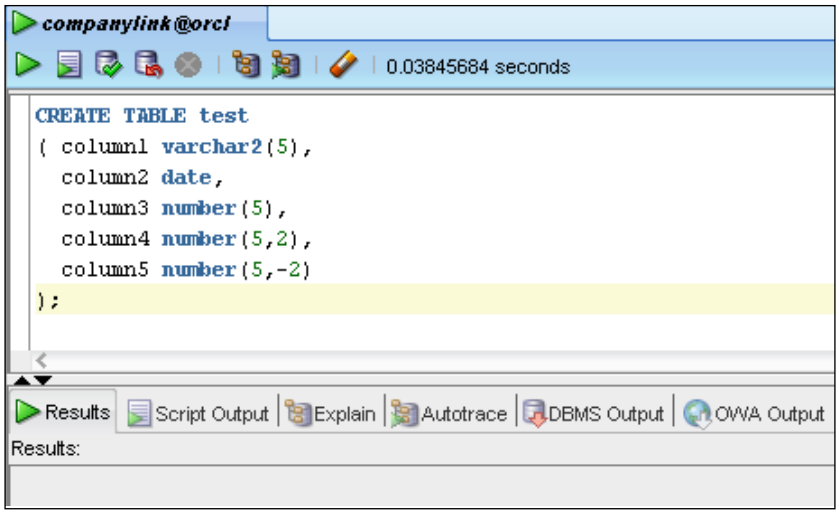


Avoiding datatype errors

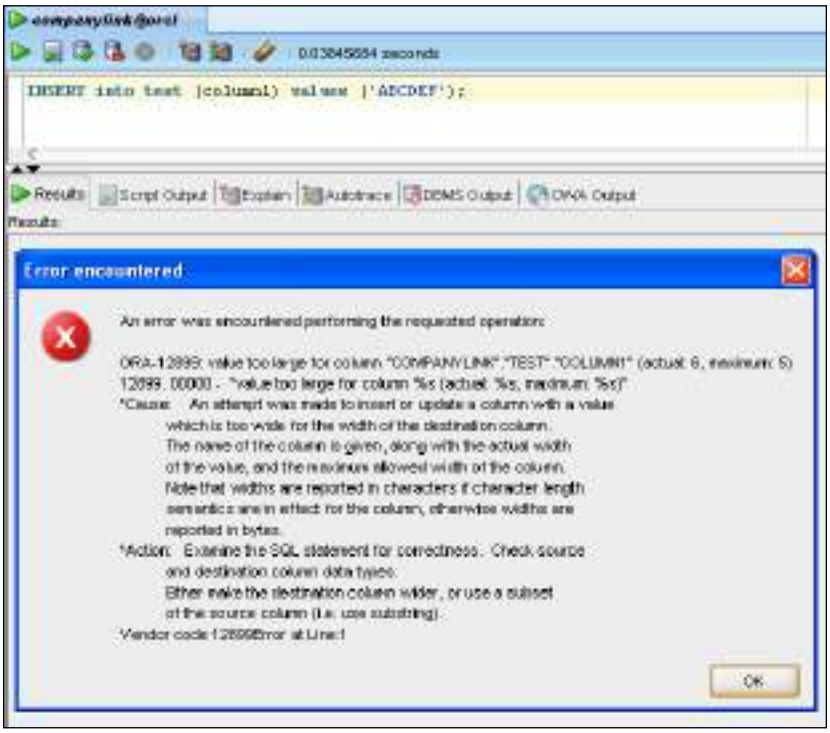
Now that we've looked at various datatypes and created a simple table, let's look at ways to avoid one of the most common types of table creation errors. This section focuses on the errors that occur due to improper usage of datatypes. Many of these errors take place because of datatype *overflow*—where values inserted into a column are too large for the scope of the declared datatype. Sometimes the cause of these errors is obvious, such as attempting to put a ten-character word into a `VARCHAR2(5)`. Other errors, particularly numeric ones, are less obvious.

Avoiding character datatype errors

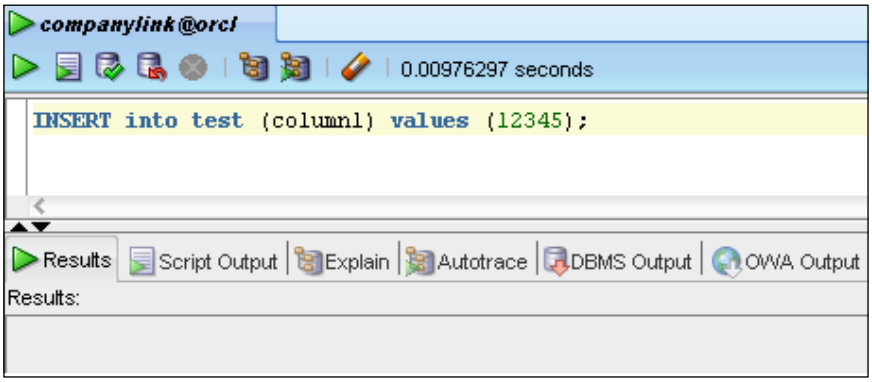
To examine the kinds of character datatype errors that are possible, let's begin by creating the table.



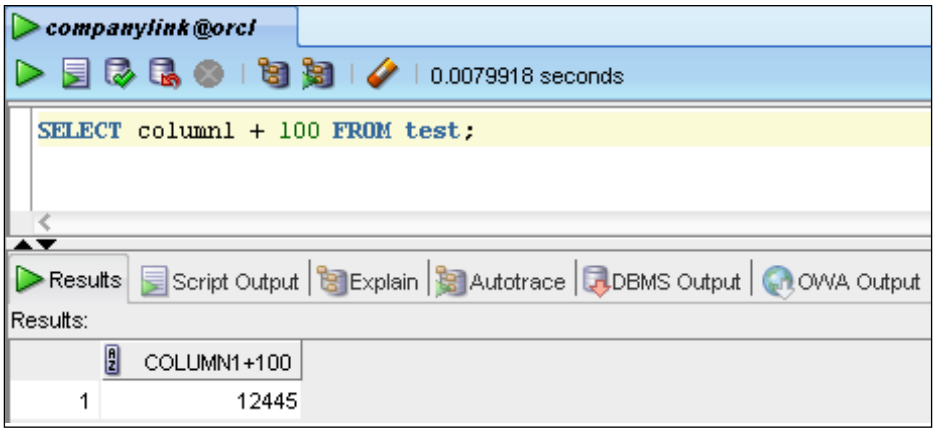
This table is created with various columns assigned with different datatypes. Some, such as the `NUMBER` types, have scope distinctions that are more subtle. We'll attempt to insert various values into these columns, some of which will generate errors. The following screenshot displays our first example of a datatype error:



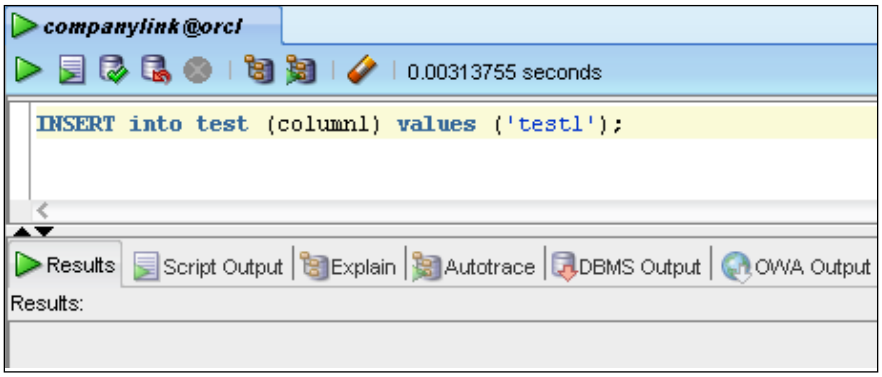
This example shows a scenario in which we've simply attempted to insert a value that is too large for our defined scope. Since the datatype in column1 is a `VARCHAR2(5)`, the six characters in 'ABCDEF' is outside the scope. We define datatypes not only on their scope, but also on their type. However, Oracle sometimes operates in ways that confuse this distinction. Note the following example:



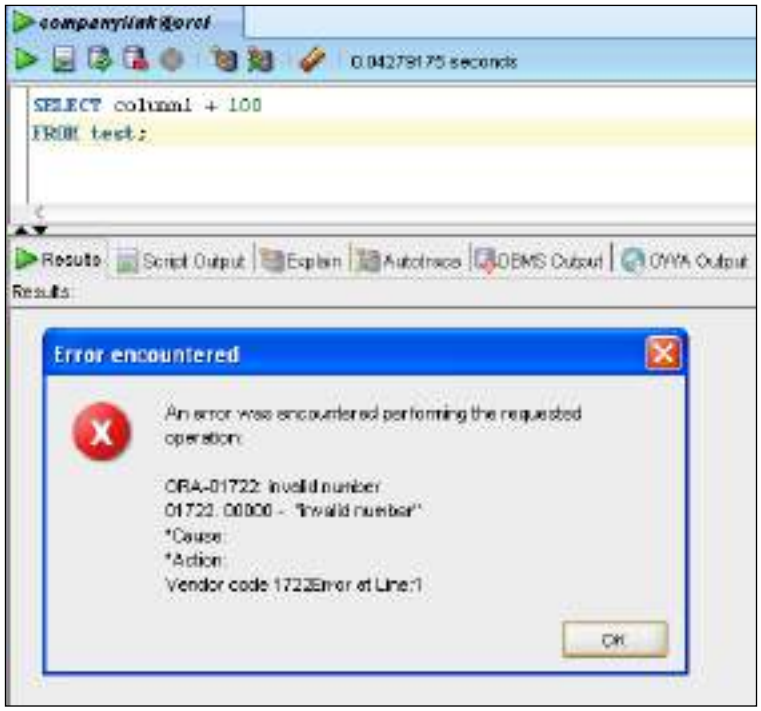
Here, we use our named column syntax for an `INSERT` statement that we learned in *Chapter 4*, *Data Manipulation with DML*. We direct Oracle to insert the value 12345 into `column1`. However, the value we inserted is numeric, while the datatype for `column1` is `VARCHAR2`—a character datatype. We didn't even use single quotes to define the value as a string. Why, then, is no error generated? Oracle will execute *implicit conversion* on some types of data when they are inserted into a different datatype. The assumption is made (and sometimes incorrectly) that since we inserted a numeric value into an alphanumeric datatype, we intended for the value to be converted. Does that mean that the value 12345 is now strictly interpreted as a string? Unfortunately, no, as the following example indicates:



As we can see, we've attempted to add 100 to our value stored in a `VARCHAR2` column. Again, no error is generated, and the correct arithmetic sum is displayed. However, the situation is further complicated if we insert a true string value into the column, as shown in the following statement:



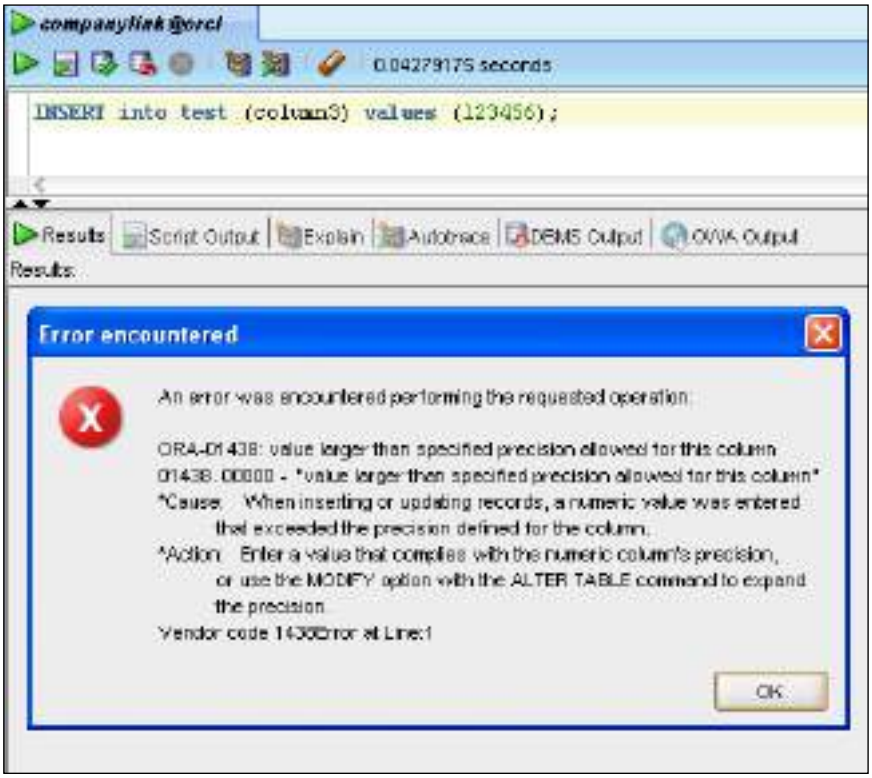
Again, since the value we insert, `test1`, is legitimate for a `VARCHAR2(5)`, the value is inserted correctly. But, if we attempt to execute the previous `SELECT` statement a second time, we see the problem with mismatching datatypes.



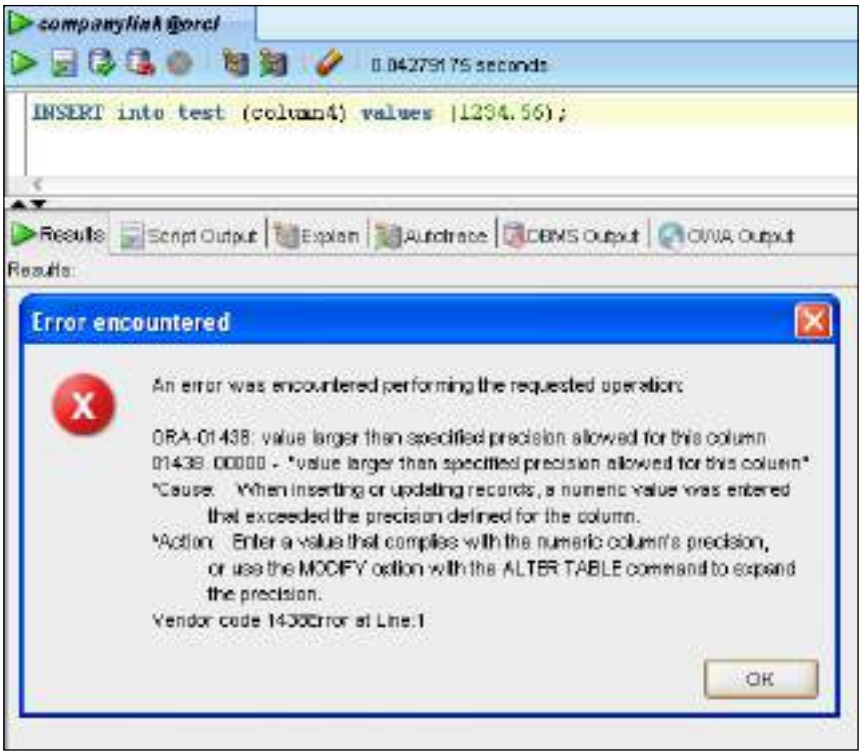
While the flexibility allowed by Oracle using implicit conversion often works in one's favor, it does underscore the importance of clearly defining column datatypes. If any kind of numeric operation or function is to be used, it is best to stay away from character-oriented datatypes.

Avoiding numeric datatype errors

To begin our look at the types of errors that can arise from numeric errors, let's look at a typical numeric overflow error, shown in the next screenshot:

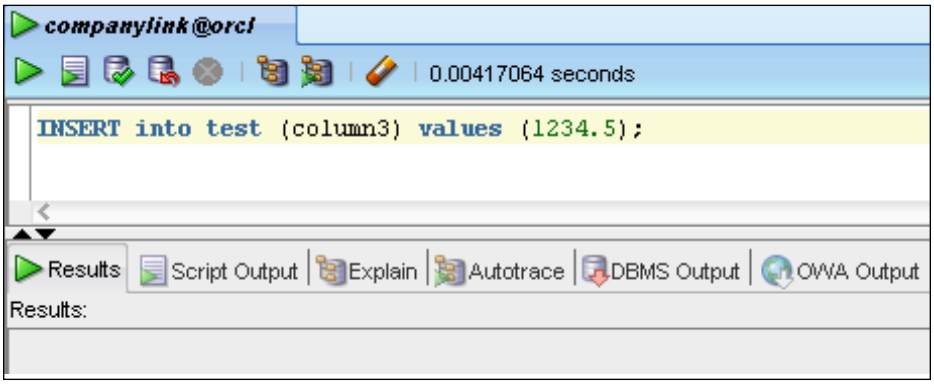


Here, we attempt to insert the value 123456 into a `NUMBER(5)`. It fails since 123456 has six significant digits, and our datatype can only hold five. We get a similar result if we attempt to place the number 1234.56 into `column4`, which has a `NUMBER(5,2)` datatype.

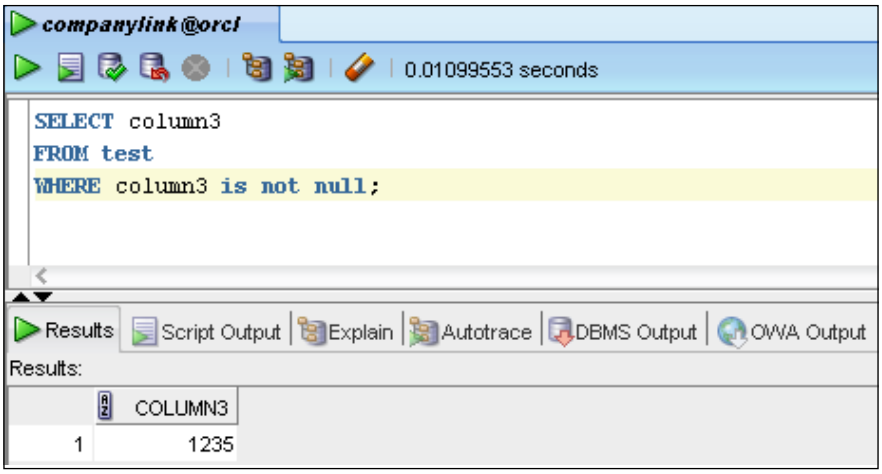


Here, even though our scale is correct (two digits), the overall number of digits we attempt to insert is six and our defined precision is five. This results in the observed error.

Although overflow errors such as these are fairly easy to interpret, there are certain errors that arise from the way that Oracle implicitly converts numeric values that are less easily observed. For instance, examine the following example:

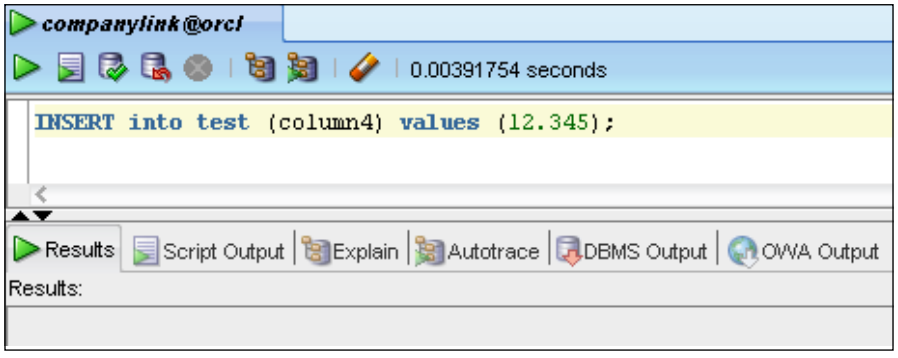


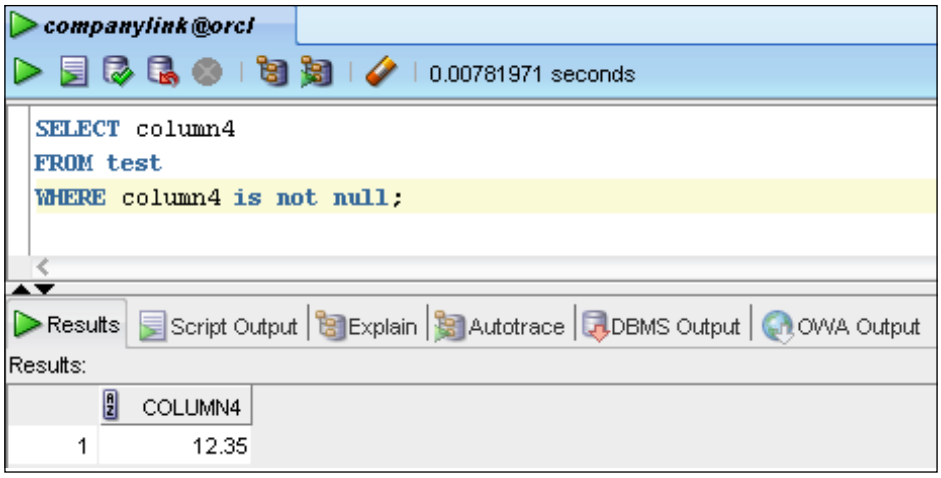
Here, we've attempted to insert a decimal numeric value, **1234.5**, into **column3,** which holds data of type `NUMBER(5)`. No scale is specified for the values in `column3`, so we might assume that an error would arise. This is not the case, as shown in the following query:



Notice that the resulting value, originally 1234.5, has been converted and stored as 1235. Not only has the decimal place been removed, but the resulting integer value has been rounded to the nearest ones place. In this case, since a `NUMBER(5)` stores values without a scale, the value is rounded to the nearest value that will fit within the defined precision. Our resulting value, 1235, fits within our established limit of 5, so no error is generated.

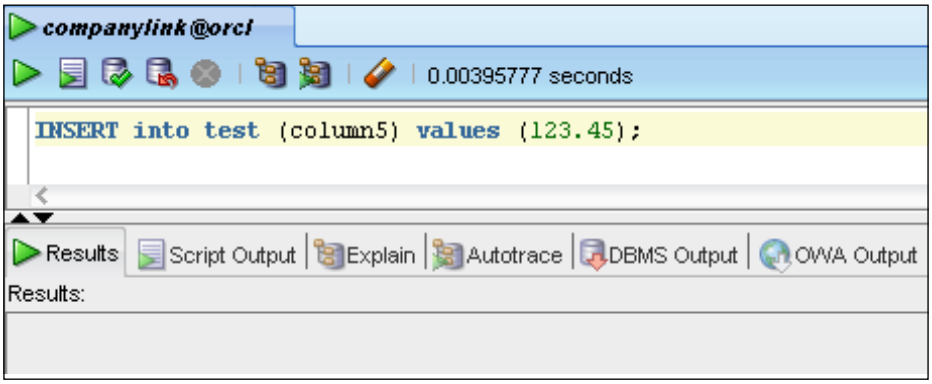
A similar conversion occurs when we attempt to insert a value that *does* fit within the defined precision, but is *greater than* the defined scale. This is shown in the next two examples.

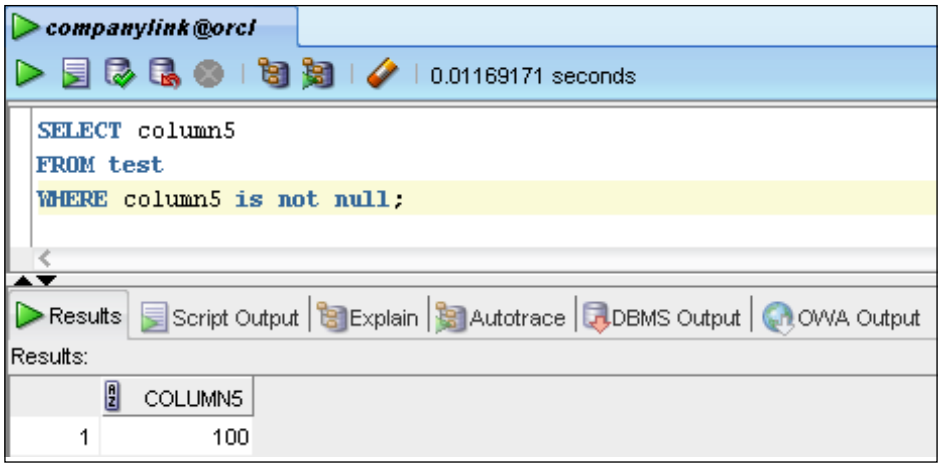




Here, we've defined `column4` datatype as a `NUMBER(5,2)`. The value we insert, 12.345, is within the precision, but has three decimal places—greater than the stated scale of two. No error is generated by this operation. Rather, Oracle rounds the number to the nearest significant digit that fits within the prescribed scale and inserts it.

Our last example of potential datatype errors looks at the behavior of values inserted into a column with a `NUMBER` datatype that has a negative scale. The `column5` column in our test table has a datatype of `NUMBER(5,-2)`. When using a negative scale, Oracle moves the significant digit by the specified scale to the *left* of the decimal point. Take a look at this behavior in the next two examples:





Although we insert the value 123.45 into column5, we see that the resulting value stored is 100. This happens because our datatype has been specified as a `NUMBER(5,-2)`. While a positive value for scale will move the significant digit to the *right* of the decimal point, a negative value moves it to the *left*. This results in the value 123.45 being rounded to a value of 100.

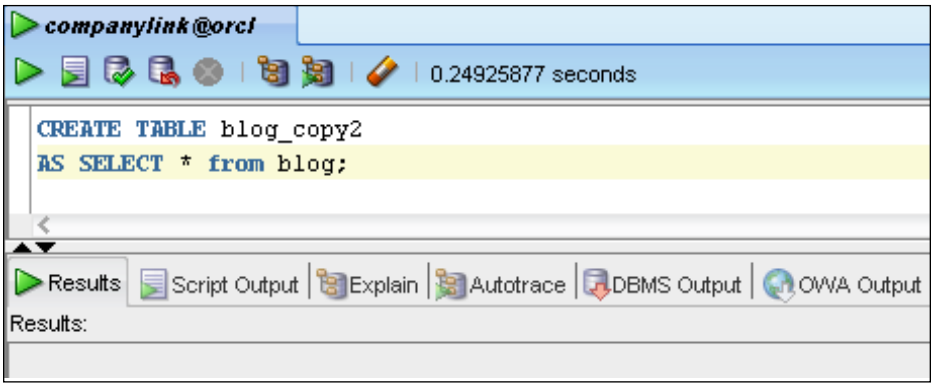
SQL in the real world

Although the variations in rounding shown previously don't necessarily constitute an error, they are often considered undesirable. If you insert the value 123.45 into a column, you generally don't want it converted to 100. Always be aware of the types of data you are using, as well as the datatypes you use to store it.

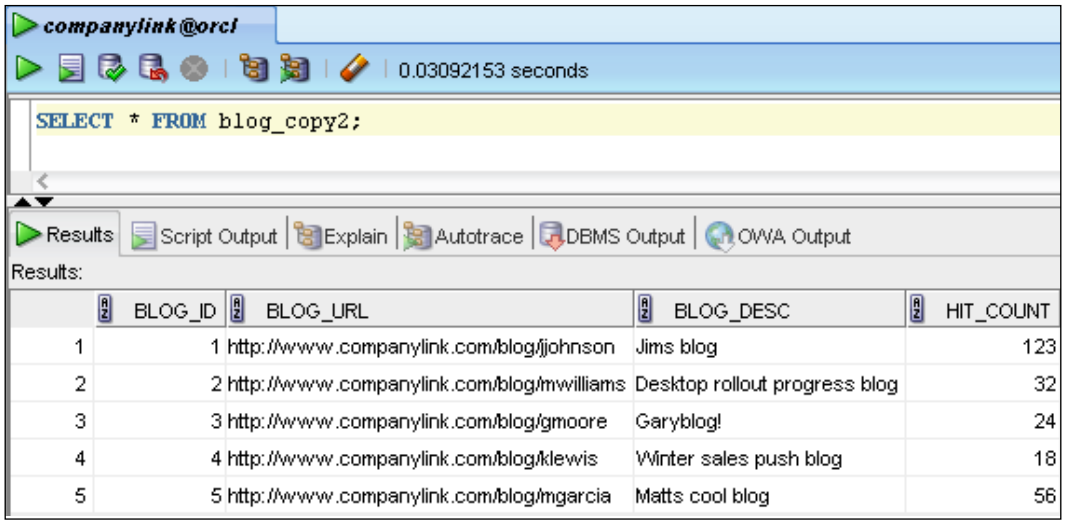
Copying tables using CTAS

In the day-to-day life of a SQL programmer, we are often called on to make a copy of a table. This could be for many reasons, such as the need for prototyping a table change or experimenting with data loads. To copy a table, we could simply run a `CREATE TABLE` statement with the same column and datatype specifications as our original and populate it with data, but this requires several steps. We would need to carefully copy the column and datatype specifications from the original. Then, we would need to create or generate numerous `INSERT` statements or load the table using another method.

Often, a better way to copy a table is by using a `CREATE TABLE... AS SELECT` statement. `CREATE TABLE... AS SELECT`, often abbreviated as CTAS (generally pronounced *see-taz*), will copy the column structure and data from an existing table in one step. An example is shown as follows:

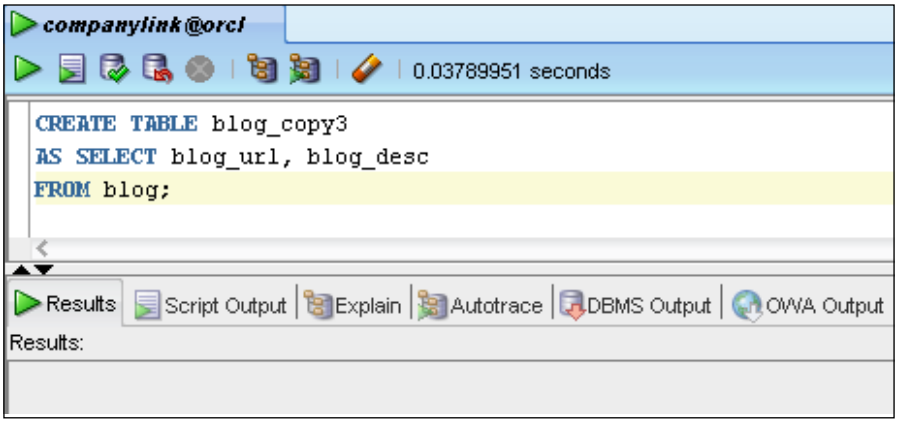


The CTAS statement uses relatively familiar syntax. Our `CREATE TABLE` and `SELECT` clauses are the same as the ones we've seen previously. The only real difference is the addition of the AS clause, which defines the statement as a CTAS. The statement reads the column structure, datatypes used, and the table data itself and creates a new table, `blog_copy2` with this information. We can see this in the following query:

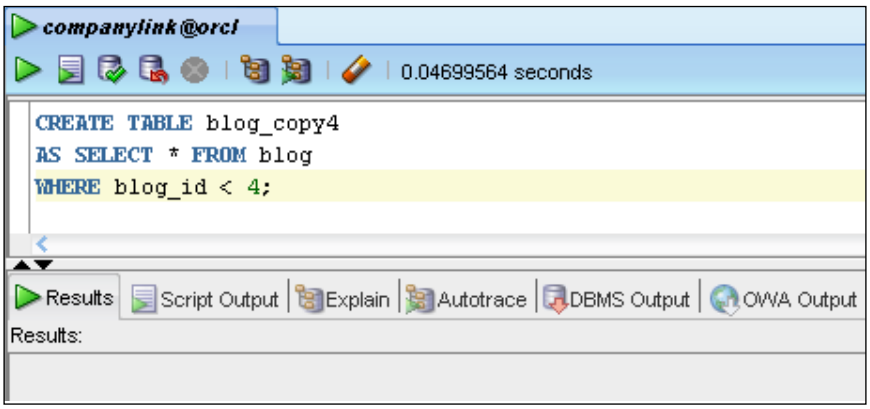


	BLOG_ID	BLOG_URL	BLOG_DESC	HIT_COUNT
1	1	http://www.companylink.com/blog/jjohnson	Jims blog	123
2	2	http://www.companylink.com/blog/mwilliams	Desktop rollout progress blog	32
3	3	http://www.companylink.com/blog/gmoore	Garyblog!	24
4	4	http://www.companylink.com/blog/klewis	Winter sales push blog	18
5	5	http://www.companylink.com/blog/mgarcia	Matts cool blog	56

The CTAS command can also be used to make a partial copy of a table, either in terms of columns or rows. The following example shows us a CTAS command that partially copies the columns from the blog table:



While the original, or source, blog table has four columns, two with datatype `NUMBER` and two with datatype `VARCHAR2`, our CTAS command only selects the two `VARCHAR2` columns. The resulting table, `blog_copy3`, contains only the `blog_url` and `blog_desc` columns, along with all the rows. This limits our copied table at the column level—we can also restrict at the row level using a `WHERE` clause.



Here, although the source table, blog, has five rows, we select only those rows where the **blog_id** value is less than **4**, resulting in a copied table, `blog_copy4`, which has all the columns from the source table but only three of the rows.

The power of the CTAS statement lies in the fact that we are only limited by the `SELECT` clause that we use. Table joins, sub-queries, single and multi-row functions, and much more can all be used as the basis for a new table. Recall for a moment some of the complex `SELECT` statements we have used through the course of the book. Any of those can be used to create an entirely new table.

SQL in the real world

Although subjects such as table permissions are outside the scope of the SQL Expert exam, it is worth noting that the CTAS doesn't make a complete copy of everything *about* the table. While a CTAS will copy the structure and data of a table, it does not copy table permissions, called grants, that allow different users to access the table.

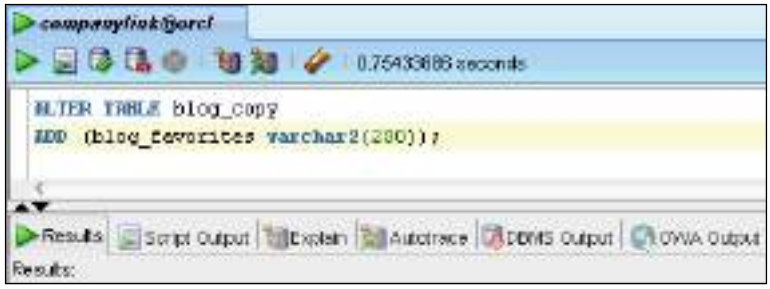
Modifying tables with ALTER TABLE

As much as we may try to make a table *perfect* the first time, SQL programmers are often called on to modify existing tables. To do so, we use the **ALTER TABLE** command. The options to modify existing tables available to us with `ALTER TABLE` are numerous. They include the ability to:

- Add columns to a table
- Remove columns from a table
- Change the storage options and location for a table
- Change the datatype and scope of columns in a table (with some restrictions)
- Rename a table

Adding columns to a table

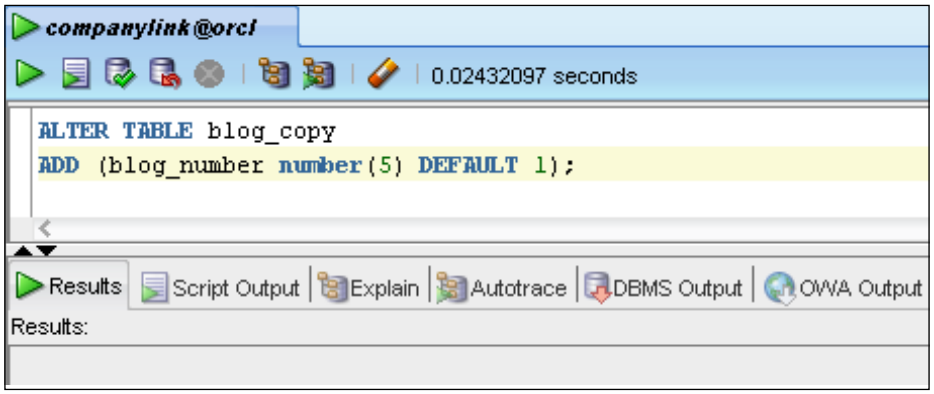
When we need to add a column to a table, we use the syntax `ALTER TABLE . . . ADD`. With this clause, we specify the name of the column to be added, along with the datatype. An example using our `blog_copy` table is shown in the next example:



In this example, we add the column `blog_favorites` to the `blog_copy` table that we created earlier in the chapter. We use the `ALTER TABLE... ADD` syntax and place the column name and datatype within optional parentheses as we do when using `CREATE TABLE`.

When a column is added to a table, that column contains no values. Thus, in our example, if the `blog_copy` table contained one million rows, the `blog_favorites` column we added would hold one million NULLs. The best way to populate columns such as these is by using an `UPDATE` statement that scans through the rows and places an appropriate value in the new column. We see the new column and its lack of a value in the following screenshot:

Another way to immediately populate a column during its addition to the table is using the `DEFAULT` keyword. In the next example, we add another column to the `blog_copy` table; a numeric one that uses the `DEFAULT` keyword:



From this `SELECT` statement, we see that the `blog_number` column has been added as the last column in the table. However, unlike our first added column, `blog_favorites`, the `blog_number` column has a value, **1**, listed in the column. The `DEFAULT` keyword automatically populates the column with the value `1` for every existing row in the table. Once a column is defined with a `DEFAULT` value, that default becomes a part of the column's behavior. Thus, when adding new rows to the table, the `DEFAULT` value will be added to that column if no value is specified. However, this behavior only works with `INSERT` statements that use named column notation. If we attempt to insert into a table using positional notation and leave no value for the column to be defaulted, we would receive an error indicating that not enough values were specified. The use of `DEFAULT` can also be specified during the creation of the table itself as part of a column's definition, immediately following the datatype definition.

SQL in the real world

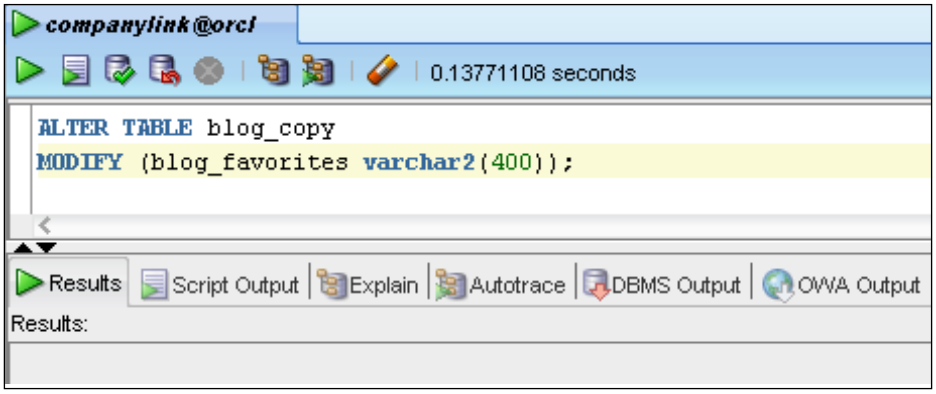
Note that adding a column with the `DEFAULT` keyword to a table with millions of rows will take a significant amount of time. One trick to avoid this problem is to add `NOT NULL` to the end to make an instant addition. For example:

```
ADD (blog_number number(5) default 1 not null);
```

We can also add the column without a default value and then add it immediately after using an `UPDATE` statement.

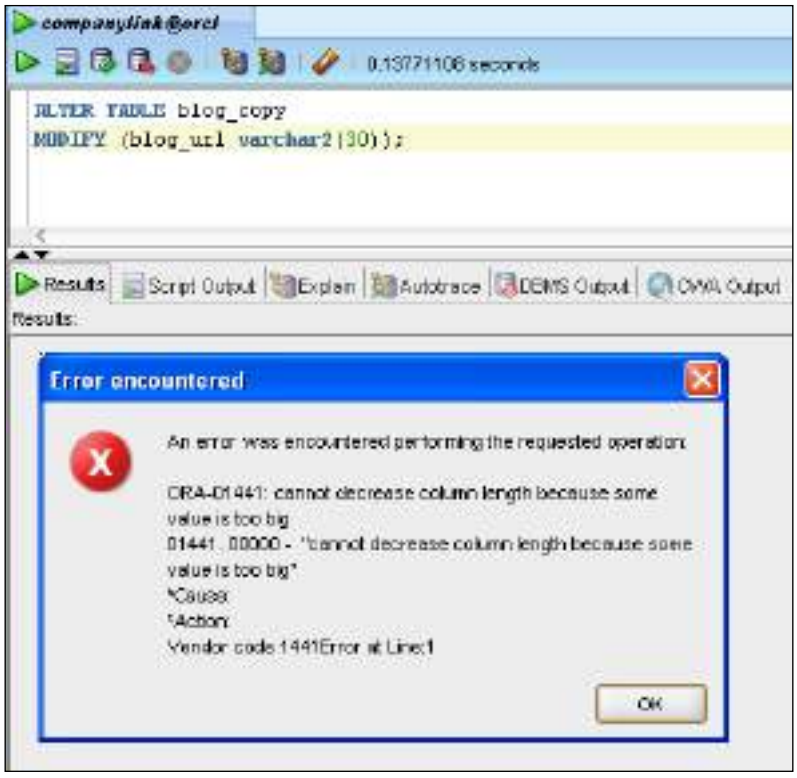
Changing column characteristics using ALTER TABLE... MODIFY

Despite our best attempts at data analysis, at times: we need to change the characteristics of existing tables. Say that we create a table with a column that contains a URL value in our `Companylink` database. Later, it turns out that the column isn't large enough to hold many of the long, complex URLs that we need to store. In this situation, we need to be able to widen, or increase the maximum size, of our URL column. To do operations such as these, we use the `ALTER TABLE... MODIFY` statement. An example of widening a column is shown as follows:

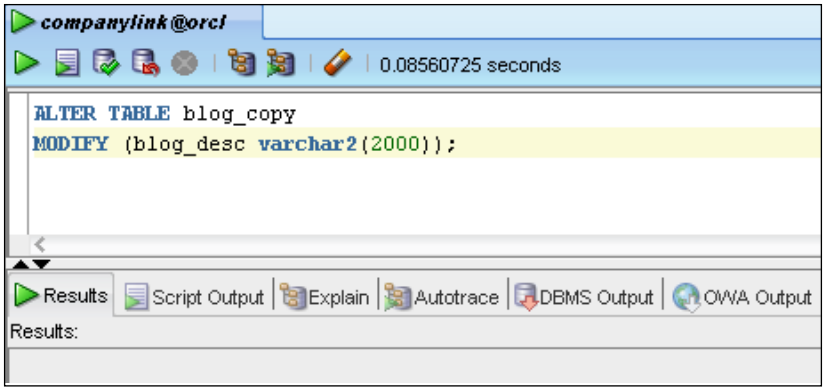


We use the keyword `MODIFY` to indicate that we are changing the specified column, `blog_favorites`, in some way. We essentially reassign the datatype and size. Here, we've changed the `blog_favorites` column in our `blog_copy` table to have a maximum length of `400` instead of the original `200`. This allows for larger values than previously. Note that this operation only changes the maximum length for values in the column and not the values themselves.

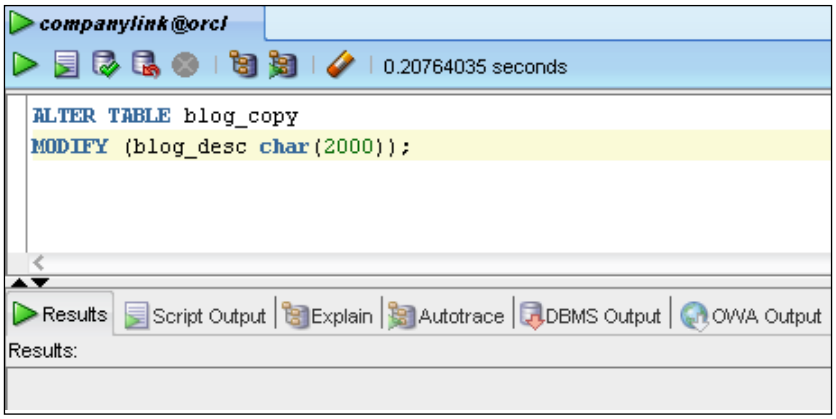
If we can use `ALTER TABLE... MODIFY` to increase the maximum length of a column, can we also use it to *decrease* that max length? We can, with certain caveats. We can decrease the max length of a column, but not less than the length of the widest value contained in the column as shown in the following example:



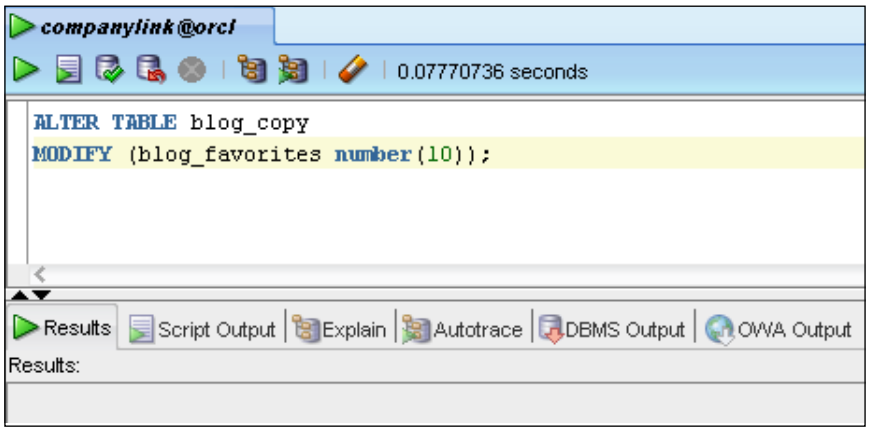
Our `blog_url` column in `blog_copy` has a maximum length of `250`. Here, we've attempted to decrease the max length to 30. However, our row in the table has a column value in `blog_url` that is `40` characters in length. Thus, we cannot resize it to a length of 30. We can, however, resize a column to a lower max length if no values in the column are greater than that value. The next example correctly resizes the `blog_desc` column from a max length of `4000` to `2000`:



We can also use `ALTER TABLE... MODIFY` to change the datatype of a column, with certain restrictions. We can change the datatypes of columns into other similar datatypes, such as `CHAR` into `VARCHAR2` and vice versa. In our next example, we convert the `blog_desc` column, a `VARCHAR2`, to a column with the `CHAR` datatype.



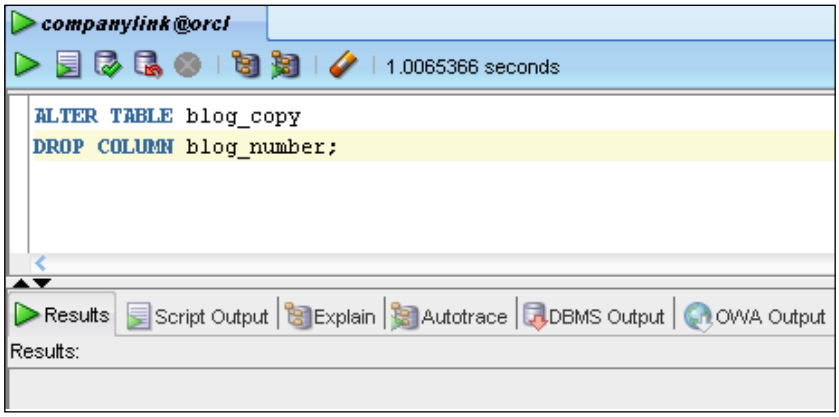
We can only change the datatype of a column to a dissimilar one if the column is empty. Our `blog_favorites` column has datatype `VARCHAR2(2000)`. However, we can change it to a numeric datatype, `NUMBER(10)`, but only because there are no values contained in it. This is shown as follows:



The ALTER TABLE... MODIFY statement can also be used to change the DEFAULT value of a column if it has been defined, or add one if it has not. However, the changes only affect future data, not the existing data in the table.

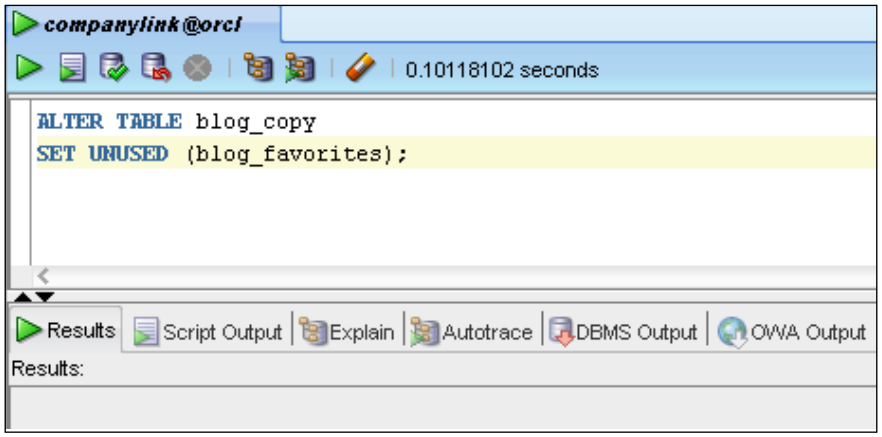
Removing columns using ALTER TABLE... DROP COLUMN

Finally, with the last of our ALTER TABLE commands, we can remove, or drop, a column altogether. To do this, we use ALTER TABLE... DROP COLUMN, as shown in the following screenshot:

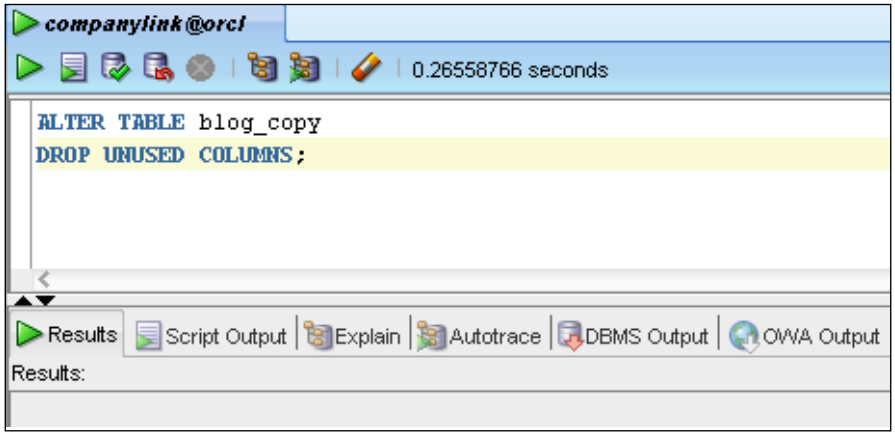


There are a few syntactical rules to remember about ALTER TABLE... DROP COLUMN. First, unlike the clauses ADD and MODIFY, this statement uses the syntax DROP COLUMN instead of simply DROP. Second, while ADD and MODIFY both enclose the specified column within parentheses, DROP COLUMN does not. We simply list the name of the column to be dropped. When a column is dropped, all of the data it contains is also dropped.

When a column is dropped, the table in question is locked, preventing certain access to it. For this reason, there is an alternative method to drop columns—one which gives the administrator greater control. Using the `ALTER TABLE... SET UNUSED`, we can "mark" a column as unused. Once a column is set as unused, it is no longer displayed as part of the table structure and cannot be used to contain new data. It is essentially made invisible. In our next example, we set the `blog_favorites` column to unused.



Once a column is set to an unused state, it can be deleted at a later time using the `ALTER TABLE... DROP UNUSED` command. This command drops *all* columns in a particular table that have been set to unused. This process allows an administrator to wait until an opportune time to drop a column from a table, such as off-hours when the table is not heavily used. In the following screenshot, we drop the `blog_favorites` column that was set to unused:

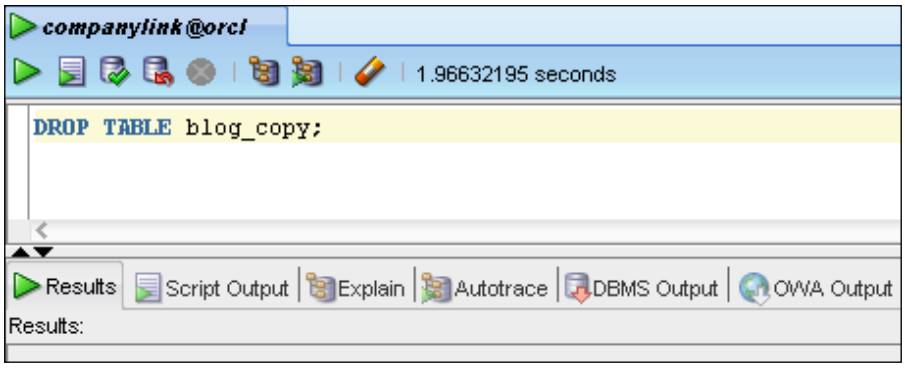


SQL in the real world

The `SET/DROP UNUSED` commands were added by Oracle to benefit large database environments with a high degree of activity, such as data warehousing and decision support environments. In such systems, dropping a column during times of high activity can produce unwanted performance issues.

Removing tables with DROP TABLE

When they are no longer needed, tables can be dropped using the `DROP TABLE` command. Unlike the `TRUNCATE` command, which removes only the data from a table, the `DROP TABLE` command removes both structure and data. Its syntax and use are shown as follows:



As with the other commands shown in this chapter, `DROP TABLE` is a DDL command. Once a table is dropped, it is dropped permanently. There is no facility that allows us to *rollback* a dropped table. We've created several test tables in this section. If you wish to remove them, you can use the `DROP TABLE` command with the following tables:

- `blog_copy2`
- `blog_copy3`
- `blog_copy4`
- `test`

SQL in the real world

Because of the irreversible nature of `DROP TABLE`, it is sometimes advisable to save a temporary copy of either the table or its data before a drop. We can use commands such as CTAS or tools such as Oracle's Data Pump to do this.

Using database constraints

Throughout this book, we have continually noted that what makes a relational database different are the relationships formed between its tables. In *Chapter 1*, *SQL and Relational Databases*, we discussed the entity relationship diagrams that visually display these relationships. We noted that in a relational model, tables may have a one-to-one or one-to-many relationship. For instance, in our Companylink data model, each employee can have one or more addresses listed in the `address` table. But, what would happen if we accidentally attempted to insert address information into the `address` table for an employee that didn't exist in the `employee` table? Up to this point in what we've learned, nothing is really stopping this from happening. However, doing so could have unforeseen consequences in subsequent queries. If the `address` table has rows that don't relate back to the `employee` table, then no relationship really exists between the two. There are other situations where the wrong type of data might be entered into a row. Datatypes are designed to prevent this from happening, but only to a certain degree. For instance, the gender column in our Companylink `employee` table should generally hold one of three possible values—'M' for male, 'F' for female, and possibly an 'N' for "not specified". What if a letter 'R' is input by mistake? We would most likely write queries that use the gender column under the assumption that the column holds one of the three possible values. If there are other values in the table, they would likely be accidentally excluded. We need a way to prevent situations like this from happening.

Understanding the principles of data integrity

We refer to rules such as these as the *business rules* for our data model. Business rules usually need a type of enforcement that goes beyond simple datatypes. Datatypes alone cannot enforce correct table relationships and the inputting of correct values. The process of enforcing business rules in a database model is known as **data integrity**. Data integrity ensures that the data contained in the database conforms to the particular characteristics of our business model. Table relationships, data integrity checks, and mandatory fields are all examples of data integrity.

SQL in the real world

The topic of data integrity can be a somewhat controversial one. Besides enforcing it at the database level, it can also be done at the application level. For instance, developers could create a web page that evaluates values for certain conditions before they are sent to the database. However, there is a place for both—it is generally a mistake to entirely exclude one method for another.

Enforcing data integrity using database constraints

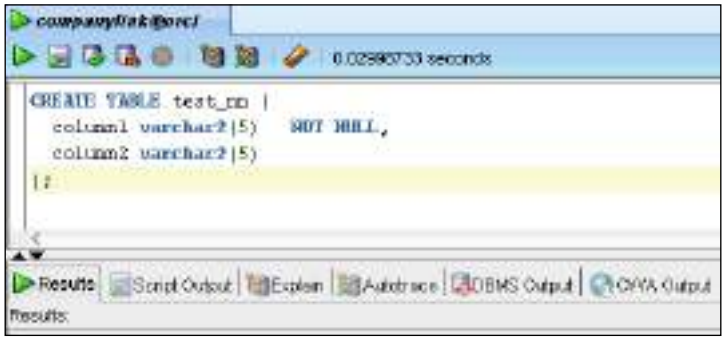
To ensure that our database data maintains the proper level of data integrity, we make use of **database constraints**. Database constraints place certain conditions on data as it is entered into our tables. There are five primary types of constraints available to us in Oracle:

- `NOT NULL`
- `PRIMARY KEY`
- `FOREIGN KEY`
- `UNIQUE`
- `CHECK`

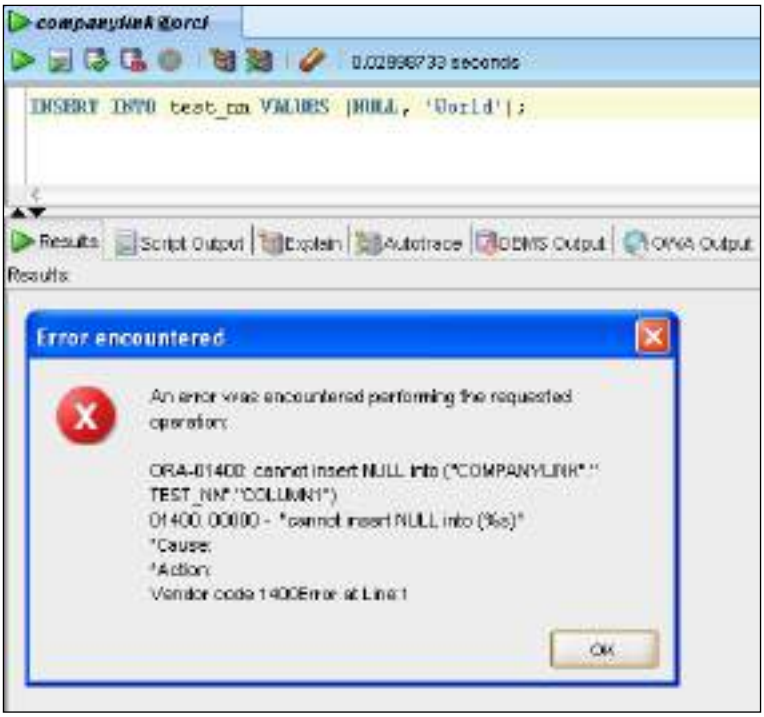
Constraints are associated with a particular column. They can also be created at two levels—either as part of the column definition or the table definition. Some, but not all, types of constraints can be done at both levels. We explore each of the constraint types in upcoming sections and note at which levels they can be created. We'll start by using constraints on small test tables and then move on to using constraints within our `Companylink` tables.

NOT NULL

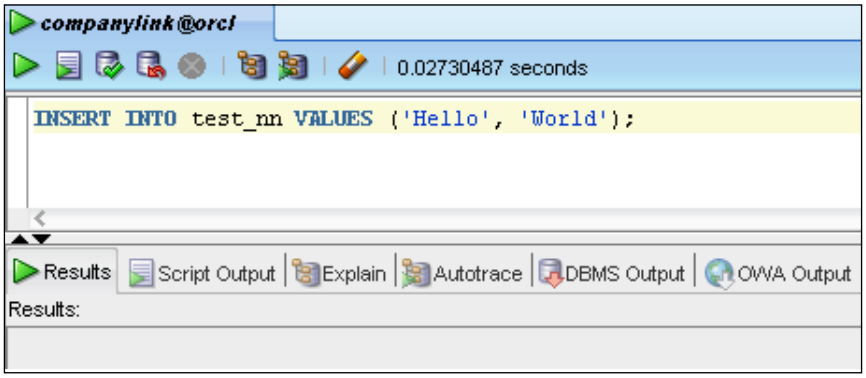
The most basic type of constraint available to us is the **NOT NULL** constraint. We've seen `NOT NULL` before, but we hadn't identified it as one of our five constraints. `NOT NULL` enforces mandatory values in our tables. If a column is constrained by `NOT NULL`, then no NULL values can be entered. A `NOT NULL` constraint is defined only at the column level, which means that it can only be created using a `CREATE TABLE` or `ALTER TABLE` statement within the column definition. The following example shows the creation of a test table with two columns—one with a `NOT NULL` constraint and one without:



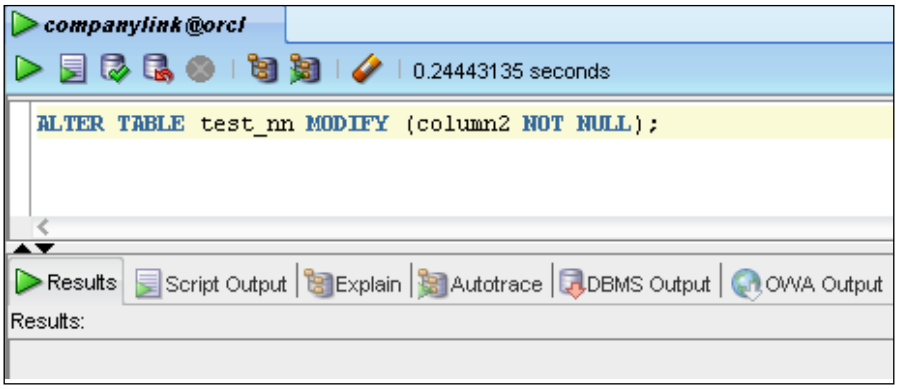
When we enter data into the `test_nn` table, our `NOT NULL` constraint prevents us from entering NULLs into `column1`. We see this demonstrated here:



If we change our insert statement, as shown in the next example, we can enter a value into both `column1` and `column2`.



As we mentioned, column level constraints can also be added to columns using an `ALTER TABLE... MODIFY` statement. To do so, we include the `NOT NULL` keyword in the column definition, similar to the way we changed datatype definitions earlier in the chapter. Adding a `NOT NULL` column constraint to a table is demonstrated here:



Note that we *cannot add a* `NOT NULL` *constraint to a column that has existing NULL values*. To do so will result in an error and would require us to update all existing NULLs with proper values before placing the constraint on the column.

PRIMARY KEY

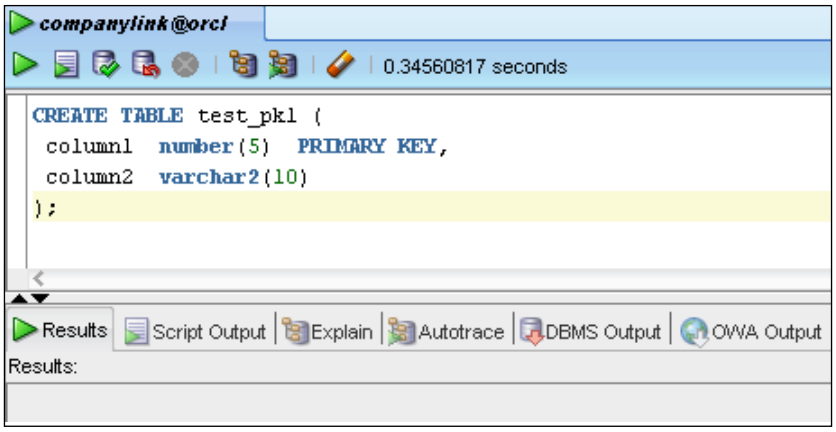
The `PRIMARY KEY` is the constraint that forms the foundation of the relational database model. When paired with a `FOREIGN KEY` constraint on another table, it forms the *parent-child* between the two. It has three characteristics that allow for the establishment of relationships between tables:

- A primary key value cannot be null
- All primary key values must be unique within a column
- Primary key values can optionally be paired with a corresponding foreign key value in another table, forming a relationship

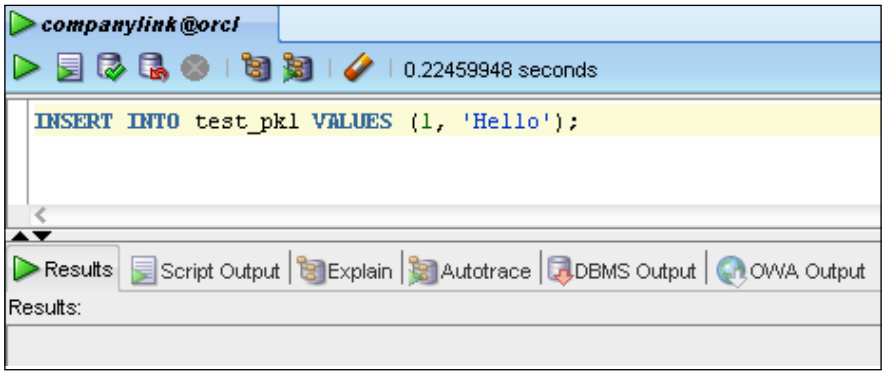
SQL in the real world

Because its values are unique, a primary key can be used to uniquely identify each row in a table. For this reason, businesses often use account numbers as primary keys. Your bank account number might well be the primary key for a table in a banking database system.

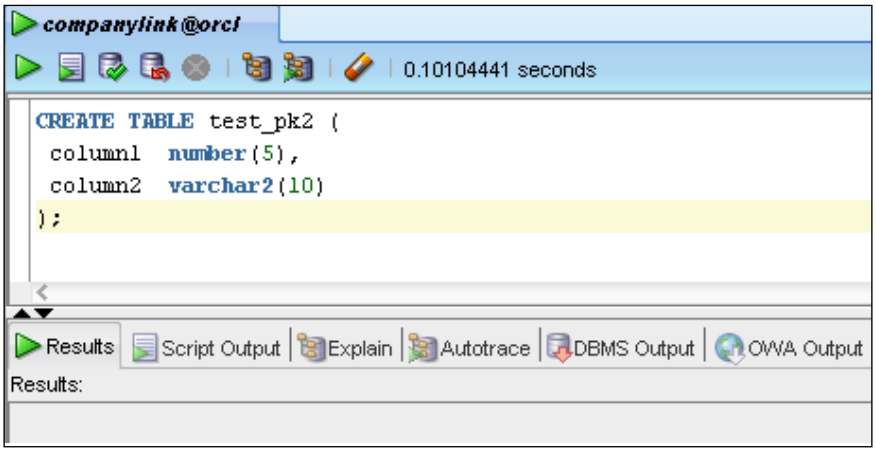
Primary keys can be defined in many different ways. A `PRIMARY KEY` can be defined on both new tables and existing tables using `CREATE TABLE` and `ALTER TABLE`, respectively. Our next statement shows us a simple test table created with a primary key:



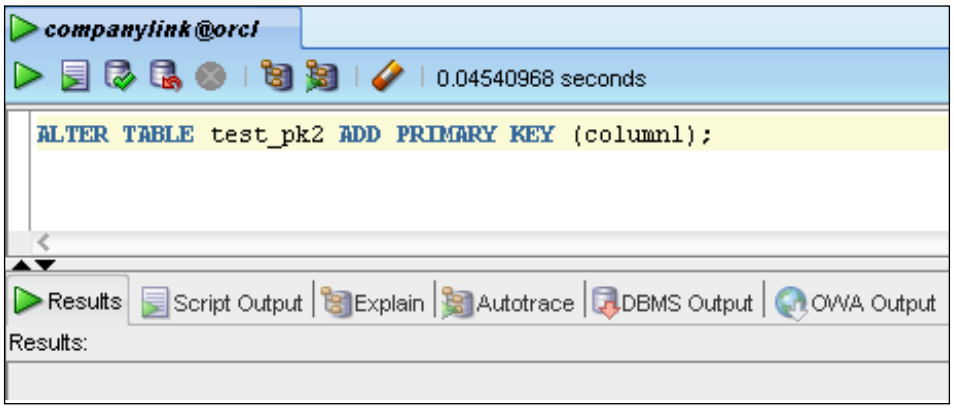
Here, the `test_pk1` table has been defined with a primary key on `column1`. From this point, values inserted into that column must be unique within the column and not null. Any attempt to insert a value that violates those constraints will result in an error.



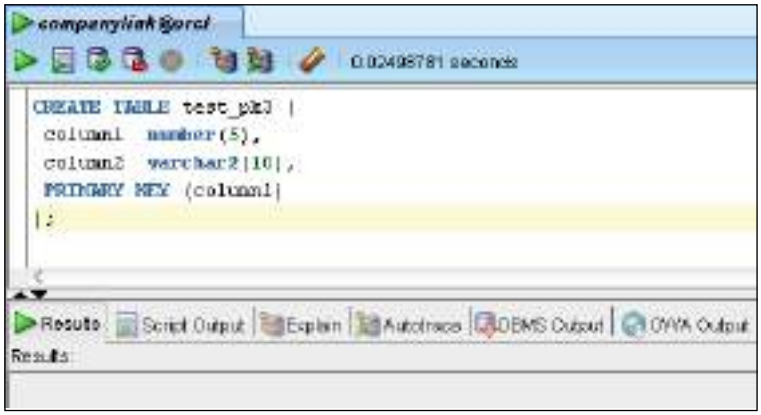
A primary key can also be added after a table is created. In our next example, we create a table that initially has no primary key:



Next, we add the primary key on `column1` using `ALTER TABLE... ADD PRIMARY KEY`.



In addition to these two ways of creating a primary key, they can be added using a third method as well. The third method uses what is known as an **out of line constraint**. The previous examples make use of **inline constraints** because they are defined with the column definition. Out of line constraints are defined with the table, but they are added in a clause near the end of the `CREATE TABLE` statement, as shown in our next example:



A table can have only one primary key. It can, however, use a primary key that is formed by more than one column. This is known as a **composite primary key**. When a composite key is used, it is the pair of values itself that must be unique. Thus, with a composite primary key, we could have multiple occurrences of the same value in one column, provided that each value is paired with a different value in the second column. It is the pair of values itself that must be unique. In the following screenshot, we create a table that uses two columns as its primary key. We do this using an out of line constraint with a slightly different syntax that allows us to actually give the primary key a name. Note that a composite primary key can only be created out of line, since inline constraints can only be used in reference to a single column.

Natural versus synthetic

In real databases, the types of data used for primary key values are handled in one of two ways. Some organizations use an actual value for the primary key, while others will generate a value. A **natural key** (also known as a domain key) is a primary key whose value actually has meaning within the business model. An example of a natural key would be a column that contains a social security number. The SSN is used to maintain uniqueness throughout the table and is actually used within the application. By contrast, a **synthetic key** (sometimes called a surrogate key) is a value that is generated and has no purpose other than maintaining uniqueness and forming relationships with other key values. A synthetic key is often generated using a database object called a **sequence** (covered in the next chapter) that generates sequential values as the data is inserted into a table. Our `Companylink` tables make use of synthetic keys. The first column of each table holds a value that exists simply to uniquely identify each row and relate that column to another table. Thus, in the `employee` table, the values in the `employee_id` column have no intrinsic meaning. They are values that can be used for a synthetic key.

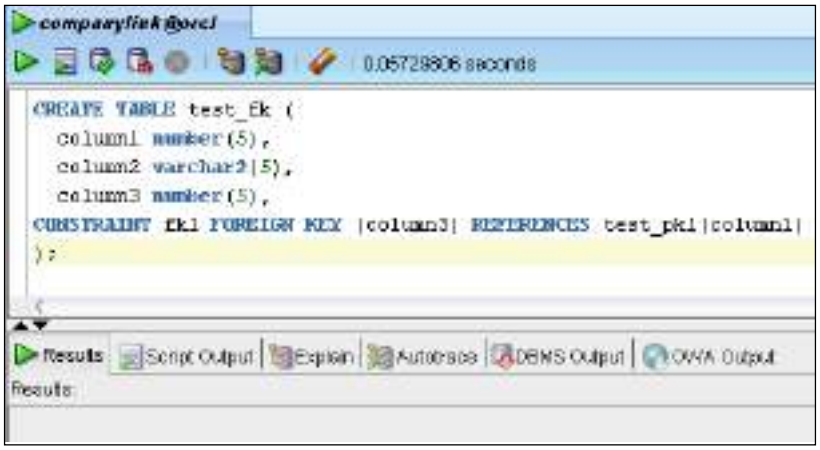
SQL in the real world

The choice to use natural or synthetic keys is often decided by an organization's coding standards. It can often be difficult to standardize on natural keys, since not all data can be used to establish uniqueness. Although using synthetic keys requires some extra space for data that isn't *real*, in the long run it can be a good way to guarantee that table relationships are maintained.

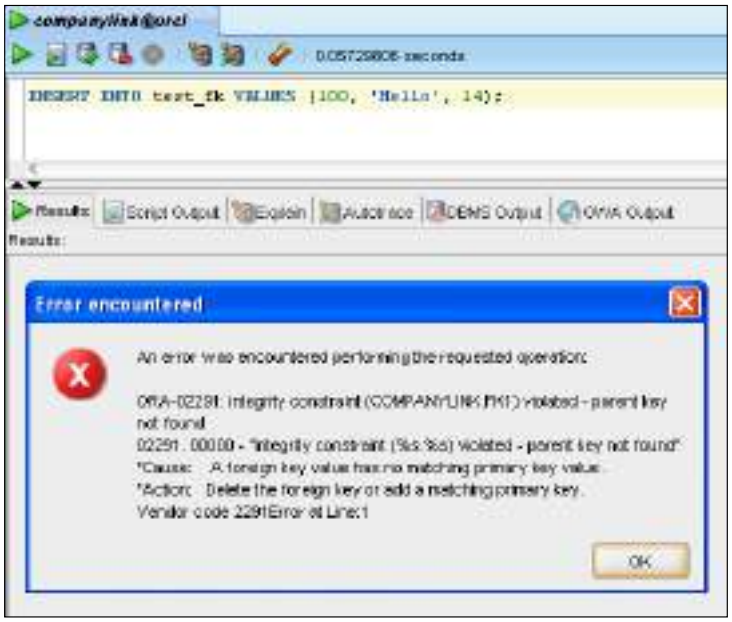
FOREIGN KEY

The complement of the `PRIMARY KEY` constraint is the **FOREIGN KEY**. A `FOREIGN KEY` forms the basis of **referential integrity**—the foundation of a table relationship. Referential integrity ensures that when a value is entered into a child table, that key value must have a corresponding key in the parent table. Thus, if we enter a value, 143, into a table with a foreign key relationship, there must be a corresponding value 143 in the parent table. Foreign keys can be created during or after table creation and must *always* reference another primary or unique key.

In the following example, we create a child table with a foreign key that relates back to the primary key of one of our test tables:



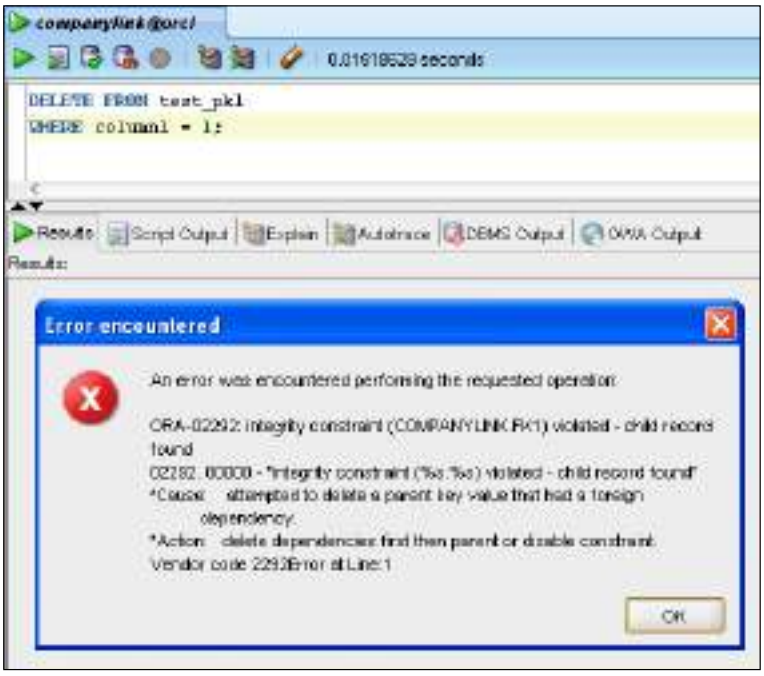
Here, `column3` in the `test_fk` table relates back to `column1` in the `test_pk1` table. We use the keyword `FOREIGN KEY` to denote the establishment of a foreign key column and the `REFERENCES` clause to indicate the table and column to which it relates. If we attempt to insert a value into `column3` that does not exist in `column1` of the `test_pk1` table, as shown in the following example, we receive an error:



If we correct this statement to include the proper values for referential integrity, our insert is successful. We insert a value of **1** into `column3`, since that value exists in our `test_pk1` table.

Deleting values with referential integrity

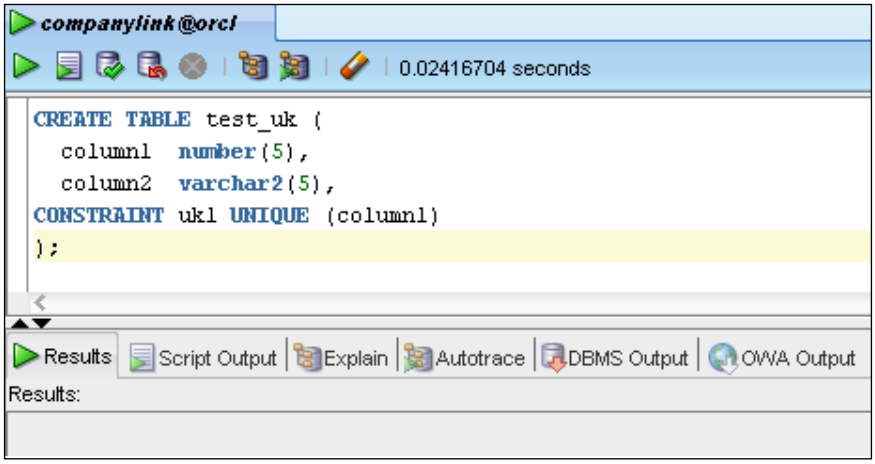
Establishing referential integrity between tables presents a unique challenge when deleting rows. We have already seen that inserting a child row with no corresponding parent row is not allowed. What about the reverse? Can we delete a parent row and leave the child row as an "orphan"? Let's test this possibility.



As we can see, we are not allowed to delete the parent record and orphan the row in the `test_fk` table. This would violate the reason that we created the relationship in the first place. In order to delete the parent row, we would need to first delete the child row that matches the parent and then delete the parent row.

UNIQUE

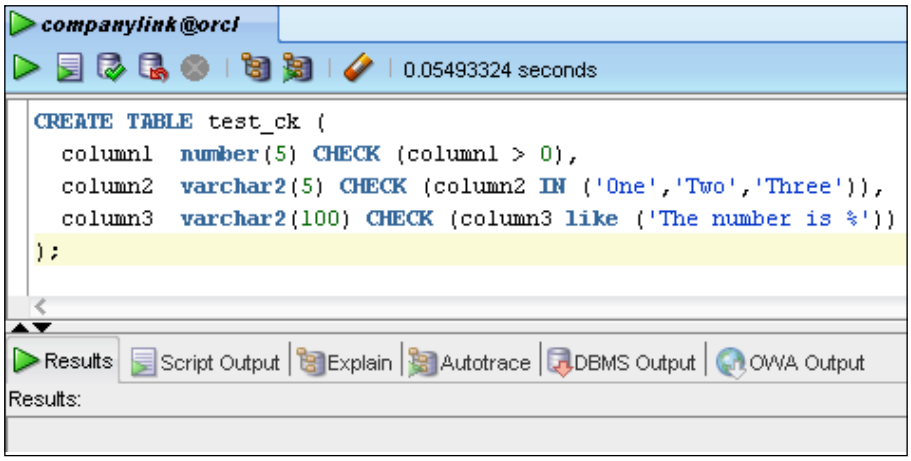
A somewhat less common constraint is the **UNIQUE** constraint. A `UNIQUE` constraint is very similar to a primary key in that it enforces unique values within a column and that it can be related to a foreign key. However, it differs in that a `UNIQUE` constraint allows NULL values. A `UNIQUE` constraint can be created both inline and out of line, and can be built during table creation with `CREATE TABLE` or after with `ALTER TABLE`. A `UNIQUE` constraint is shown in our next example:



CHECK

One of the primary focuses of data integrity is ensuring that proper data is entered into columns. As an example, we referred earlier in the chapter to a gender column whose business rules dictate that only the values 'M', 'F', and 'N' should be allowed. To enforce rules such as these, we use the **CHECK** constraint. When using a `CHECK` constraint, we specify a condition that must be met for the data to be allowed into the table. These conditions are similar to the conditions for `WHERE` clauses and often include conditions of equality or inequality, as well as the `IN` and `LIKE` clause. `CHECK` constraints can be created with the table or added later with `ALTER TABLE`. They can also be created inline or out of line. However, any `CHECK` constraints that involve multiple columns must be defined out of line.

Our next statement shows a table created with several `CHECK` constraints:



The example indicates that `column1` can only accept values that are greater than zero, `column2` can only accept the values **'One', 'Two',** or **'Three'**, and `column3` must include the string, **'The number is'** at the beginning of every value.

Extending the Companylink Data Model

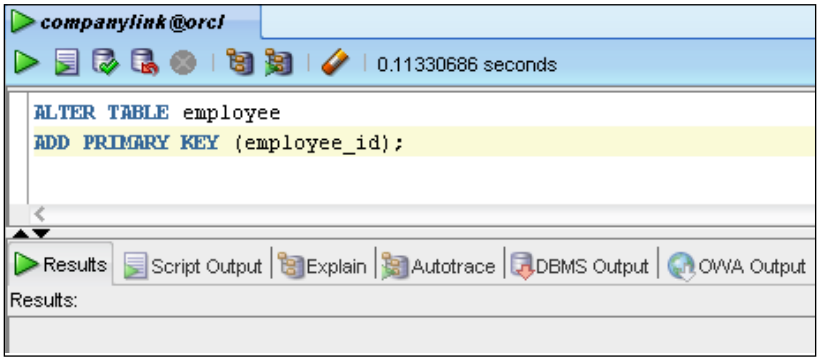
Thus far, we've learned about the syntax and use of constraints using simple test tables. In this section, we take what we've learned and apply it to creating new tables and constraints in our `Companylink` database.

Adding constraints to Companylink tables

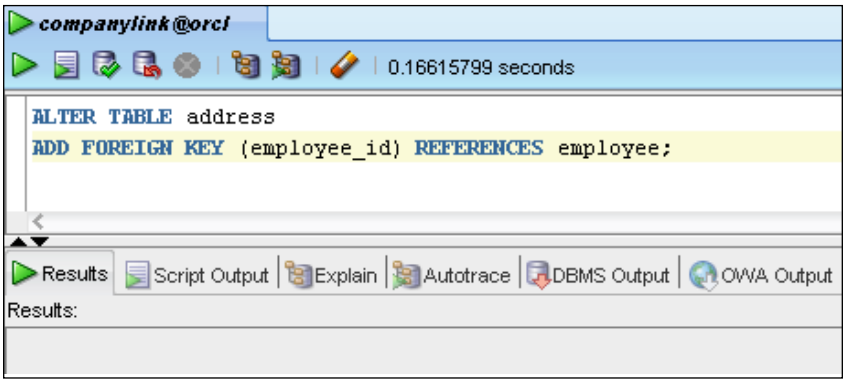
Our `Companylink` tables have been designed as a fully relational model. Each table has an ID column that relates to another table, although no constraints exist. We can take what we've learned thus far and add constraints to our `Companylink` tables. Our model is centered on our `employee` table, so we can start there.

Adding referential integrity

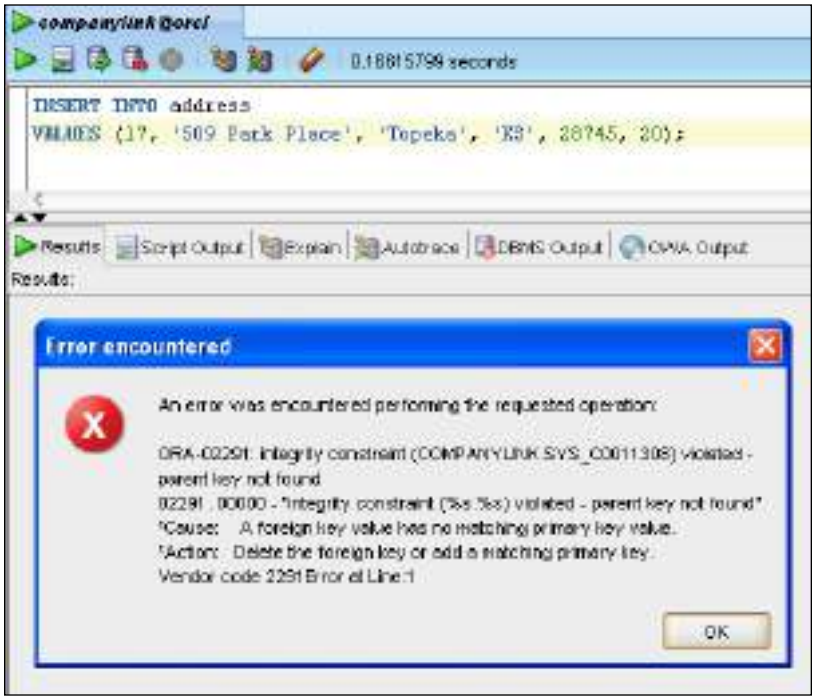
In this example, we add a primary key to our `employee` table:



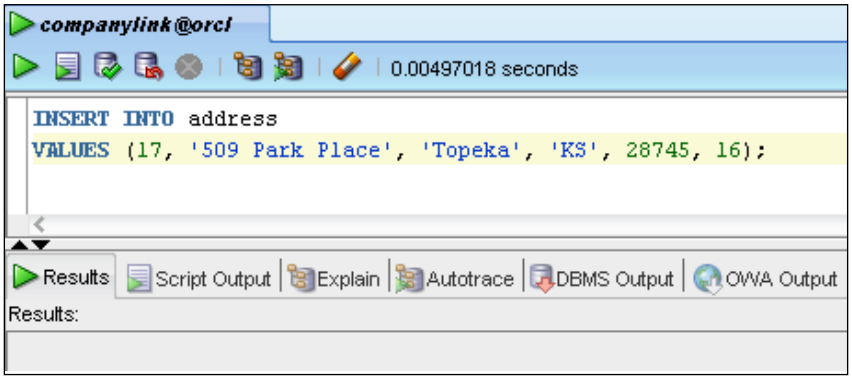
Our existing values in the `employee_id` column meet the requirements of a primary key—they are unique and no NULLs exist in the column. Next, we add a foreign key to the `address` table that relates to the `employee` table.



Once this relationship is established, we can no longer insert values into the `address` table unless a corresponding value exists in the `employee_id` column in the `employee` table. Our next statement displays an example of an unsuccessful attempt:



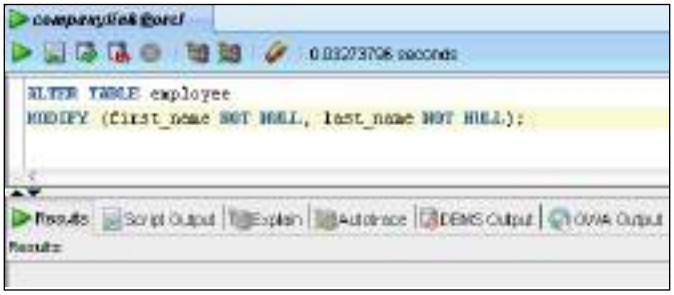
By simply changing the value inserted for `employee_id`, we can correct this problem, as shown in our next example. This relates the value back to an existing `employee_id` in the `employee` table and results in giving the employee with ID number 16, Laura Thomas, two addresses in the database. This is allowed since there is a one-to-many relationship between the `employee` and `address` tables.



The code file contains a script called `companylink_constraints.sql` that creates the primary key and foreign key relationships for the `Companylink` tables. If you wish, you can copy-and-paste the SQL statements from that file or run it as a script to create the remainder of the constraints. Remember that a relationship between `employee` and `address` has already been established in the previous examples. If you have already run those statements, you do not need to run them again.

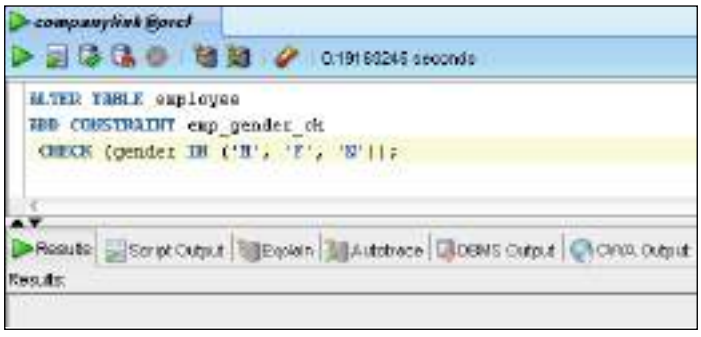
Adding a NOT NULL constraint

As people sign up for `Companylink`, we want to ensure that we record their first and last name in our database. To enforce this business rule, we can use a `NOT NULL` constraint. In the next example, we place a `NOT NULL` constraint on the `first_name` and `last_name` columns in the `employee` table.



Adding a CHECK constraint

As our `Companylink` database grows, we want to prevent incorrect data from being accidentally entered. As we've noted, we could do this at the application level. This can often involve a lot of additional coding on the application side. We, however, have chosen to do this with constraints. The next screenshot displays a simple example of a named `CHECK` constraint that limits the possible values in the `gender` column in the `employee` table to three:



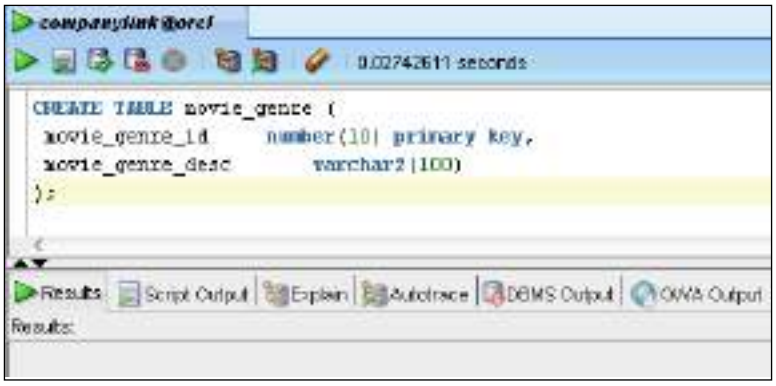
Since we're familiar with the process of adding constraints to our `Companylink` tables, feel free to try some of your own. Just be certain that constraints make logical sense and don't violate the existing table structure.

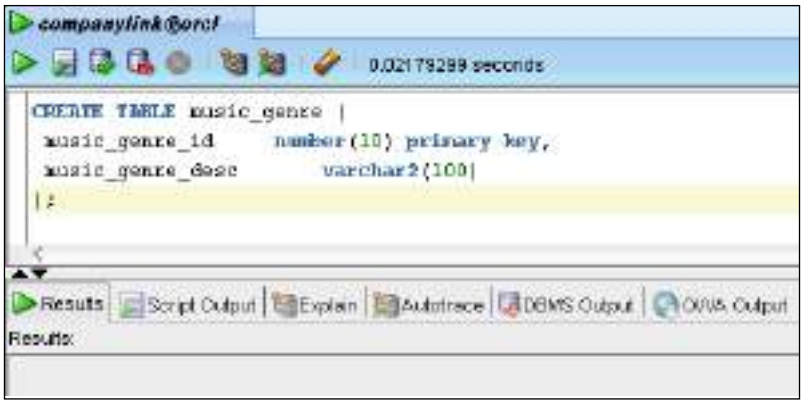
Adding tables to the Companylink model

Now that we've learned how to create tables that conform to business rules, let's expand our database. Say that we've received new requirements from the application design team. Developers are coding new changes to the `Companylink` application that will allow participants to list their personal interests on their home page. At this point, we want to add their favorite movie and music genres to our site. The developers are handling the application coding, so it's our job as SQL coders to expand our model to include this new data and create some new tables to store it.

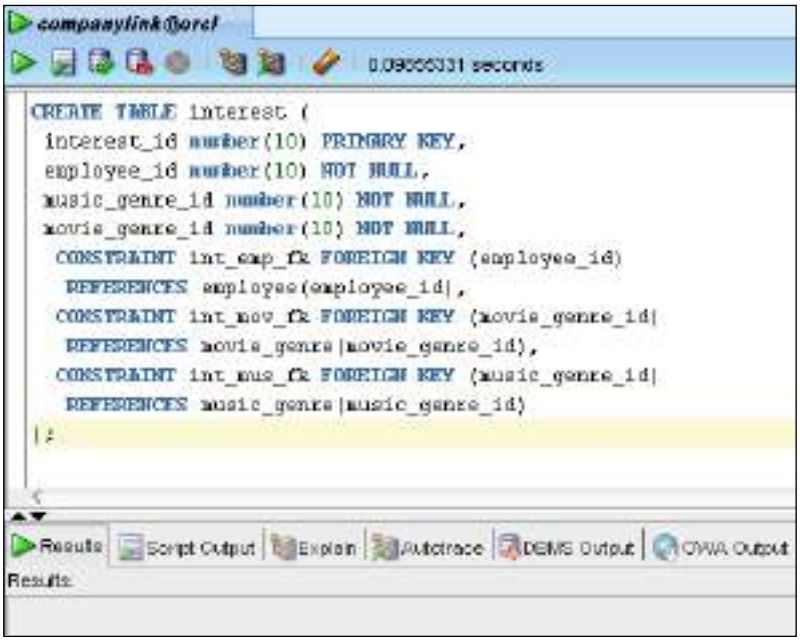
It's worth noting that there is more than one way to proceed. Employee interests could be lumped together in a single table that relates back to the `employee` table. However, as our database grows, it is possible that this wouldn't represent a fully normalized model. We could also add new columns to the `employee` table, which would be a normalized approach. But, in order to fully utilize our new table and constraint creation abilities, we'll take the approach of adding a table for interests that relates back to employee with child tables for the different interests themselves. This allows us to add a new interest down the road by simply adding another table rather than restructuring.

Our table for employee interests, called interest, will serve as a kind of *bridge* table to the specific interests. We will need to relate the interest table to the different specific interest tables, which we'll call `movie_genre` and `music_genre`. Thus, we need three tables that relate in the ways we've noted. One very important fact to remember is that the foreign key constraints will be on the interest table. Since the foreign keys can only relate to existing tables, *we must create the interest table last*. The `movie_genre` and `music_genre` tables need to exist first. We create these in the next two examples:

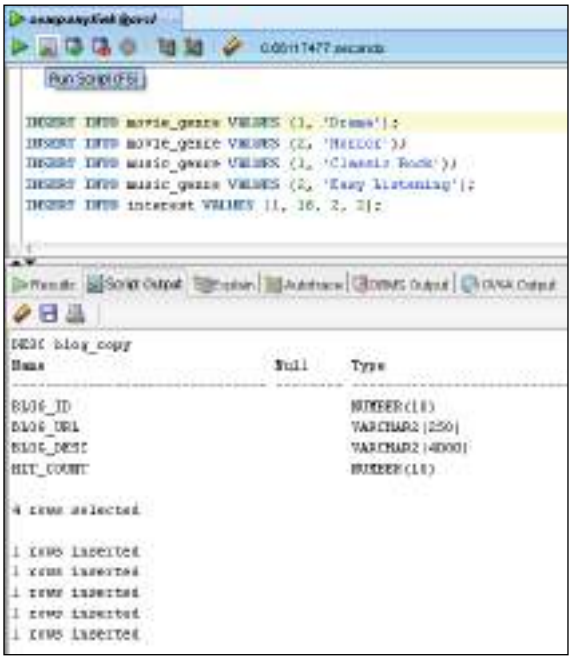




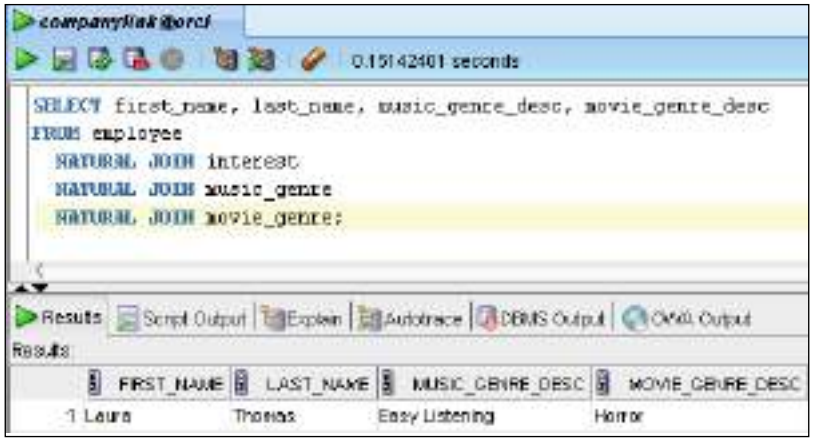
Each table has a primary key that will be used to relate to the interest table, as well as a description column that details the particular genres in each table. Once the genre tables are created, we can create the interest table as follows:



To see this in action, let's load some data for employee #16, Laura Thomas, as shown in our next set of statements. We first load a few rows for movie and music genres, then a row in the interest table with her employee ID. We can execute these separately, or together using the **Run script** button. The five inserts are shown at the bottom of the screenshot:



Now, if we want to know what Laura's music and movie interests are, we simply join the four tables, as shown in the following screenshot:



It appears that Laura likes easy listening music and horror movies—an interesting combination!

Summary

In this chapter, we've brought the key concepts of an RDBMS together with the power of SQL. Using Data Definition Language (DDL), we've learned to create tables with different datatypes that specify the kinds of data we wish to store. We've enforced business rules with different types of database constraints, including primary and foreign keys, unique keys, not nulls, and check constraints. Finally, we brought this knowledge into the real world by applying constraints to our `Companylink` tables, as well as creating new ones.

Certification objectives covered

- Categorize the main database objects
- Review the table structure
- List the datatypes that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

In this chapter, we introduced the concept of database object creation using Data Definition Language. However, there are many other types of database objects available to us. In our next chapter, we'll look at several of the most important ones, including objects that help increase database performance.

Test your knowledge

1. Which of the following statements about SQL sub-languages is true?
    a. Data Definition Language (DDL) is a sub-language of Data Manipulation Language (DML)
    b. DML is a sub-language of DDL
    c. Both DML and DDL are sub-languages of SQL
    d. Neither DML nor DDL are related to SQL

2. Which of the following terms describes a way to classify the types of data that are possible in a particular column?
   a. Formula
   b. Sub-query
   c. Function
   d. Datatype

3. Which of the following datatypes stores alphanumeric data in a fixed length format?
   a. CHAR
   b. VARCHAR
   c. VARCHAR2
   d. DATE

4. Which of the following datatypes stores alphanumeric data in a variable length format?
   a. CHAR
   b. NUMBER
   c. VARCHAR2
   d. DATE

5. Which of the following datatypes could store the value "Model Airplane" without an error?
   a. VARCHAR2(15)
   b. VARCHAR2(12)
   c. CHAR(12)
   d. NUMBER(15)

6. Which of the following numeric datatypes could NOT store the value 8479.34 without generating an error?
   a. NUMBER(6,2)
   b. NUMBER(5,2)
   c. NUMBER(8)
   d. NUMBER(8,3)

7. Which of the following lines from a `CREATE TABLE` statement will cause an error?
   a. CREATE TABLE 7_blog (
   b. new_blog_name varchar2(10),
   c. old_blog_name varchar2(10),
   d. old_blog_id number(4));

8. Which of the following lines from a `CREATE TABLE` statement will cause an error?
   a. CREATE TABLE select (
   b. first_number number(10),
   c. second_number number(10),
   d. average number(10,2));

9. Given the table creation statement below, which of the listed `INSERT` statements will execute without error?

```
CREATE TABLE car (
  make varchar2(8),
  model varchar2(15),
  vin varchar2(10),
  year number(4));
```

   a. INSERT INTO car VALUES ('Chevrolet', 'Camero', 'YA4JFI84PO', 1979);
   b. INSERT INTO car VALUES ('Ford', 'Mustang', 'JN1JFI48KD', 2010);
   c. INSERT INTO car VALUES ('Toyota', 'Land Cruiser Prado', 'BM7JPL23AQ', 2002);
   d. INSERT INTO car VALUES ('BMW', 'E81', 'RT1OUI55KD', '01-JAN-05');

10. Refer to your `Companylink` tables. After the following command is run, how many columns of each datatype exist in the "`employee_copy`" table?

```
CREATE TABLE employee_copy
AS SELECT employee_id, branch_id, project_id
FROM employee;
```

    a. 4 NUMBERs, 3 VARCHAR2s, 1 CHAR and 4 DATEs
    b. 4 NUMBERs
    c. 3 VARCHAR2s
    d. 3 NUMBERs

11. Which of the following `ALTER TABLE` statements would execute without error?
    a. ALTER TABLE blog ADD COLUMN (blog_number number(5));
    b. ALTER TABLE employee ADD employee_number number(5);
    c. ALTER TABLE project ADD (project_number number5);
    d. ALTER TABLE award ADD (award_number number(5);

12. Assuming a column, `blog_number`, exists in the blog table, which of the following modifications to the blog table is syntactically correct?
    a. ALTER TABLE blog MODIFY (blog_number datatype number(5));
    b. ALTER TABLE blog MODIFY (blog_number number(5));
    c. ALTER TABLE MODIFY (blog_number number(5));
    d. ALTER TABLE blog MODIFY COLUMN (blog_number number(5));

13. Given a table, profile, with a column called `profile_id` that has a datatype of number(10) and contains no data, what would be the outcome of the following statement?

```
ALTER TABLE profile MODIFY (profile_id varchar2(4000));
```

    a. The statement will error since a number database cannot be changed to a varchar2
    b. The statement will error since a length of 4,000 is beyond the maximum for the varchar2 datatype
    c. The statement will not error but the datatype will not be changed to varchar2
    d. The statement will successfully change the profile_id column from number to varchar2

14. Which of the following is not a type of database constraint?
    a. PRIMARY KEY
    b. SEQUENCE
    c. FOREIGN KEY
    d. CHECK

15. Which term describes a key value that is generated by the database and has no relevance to application data, other than maintaining relationships in the database?
    a. Natural key
    b. Synthetic key
    c. Sequence key
    d. Composite key

16. Which of the following statements can be used to establish a primary key?
    a. CREATE TABLE test (col1 number primary key);
    b. CREATE TABLE test (col1 number, CONSTRAINT test_pk PRIMARY KEY (col1);
    c. ALTER TABLE test ADD PRIMARY KEY (col1);
    d. All of these statements can be used to establish a primary key

17. Assume you have two tables, `test1` and `test2`. The `test1` table has a primary key. The `test2` table has a foreign key that relates back to `test1`'s primary key. Given the following statement, what happens?

```
DELETE FROM test1;
```

    a. All values are deleted from the test1 table
    b. All values are deleted from the test1 and test2 tables
    c. Zero values are deleted from the test1 and test2 tables
    d. An error occurs

18. How does a `UNIQUE` constraint differ from a `PRIMARY KEY`?
    a. A UNIQUE constraint enforces unique values, while a PRIMARY KEY does not
    b. A UNIQUE constraint cannot be used as the basis for a FOREIGN KEY relationship
    c. A UNIQUE constraint allows NULL values, while a PRIMARY KEY does not
    d. A UNIQUE constraint does not allow NULL values, while a PRIMARY KEY does

10
Creating Other Database Objects

When one thinks of the types of objects in a database, tables are the ones that most commonly spring to mind. However, there are many types of objects that can be created and stored in an Oracle database. They range from the tables that we have seen thus far to objects that can be used to speed performance or simplify queries. In this chapter, we examine a number of these objects and how they can be used within the Oracle database.

In this chapter, we shall:

- Discuss the types of indexing available in Oracle
- Examine the syntax of CREATE INDEX
- Understand the use of views and their syntax
- Look at how sequences are used to generate key values
- Examine public and private synonyms

Using indexes to increase performance

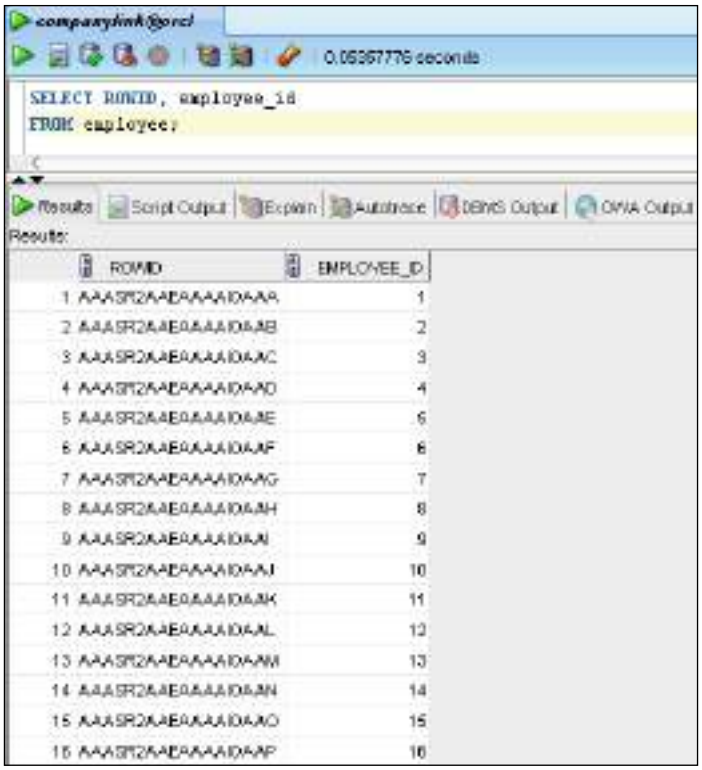
For demonstration purposes, our `Companylink` tables are very small. This is so we can easily load and manipulate them. However, in the IT industry, database tables can be enormous. Today, tables with millions of rows are commonplace, while billions of rows are not unheard of. In such environments, *database performance* becomes a serious issue. It isn't enough that your table join completes without error—it must often complete within a certain period of time (usually, *as fast as possible*). Oracle performance tuning is an extraordinarily complex subject that, for the purposes of this book, is mostly out of scope. However, since simple things can often make a significant difference, we introduce the subject of indexing, which can have a profound impact on the performance of our queries.

Scanning tables

When a query is issued to the database, a **table scan** is performed. A table scan is used to search table rows for those that match our query's conditions. When we issue a `SELECT` statement without a limiting condition in a `WHERE` clause, a **full table scan** is used to retrieve every row. We've requested every row, so it is preferable that every row be scanned. However, more often, we don't want every one of our queries to bring back every row, every time. Normally, we are attempting to bring back a row or set of rows that meet a certain condition. In such situations, it seems inefficient to search through every row in the table just to return a few. We need a way to more quickly search through a table for the rows we require.

Understanding the Oracle ROWID

In the previous chapter, we mentioned that a primary key value will uniquely identify any single row in a table. If we want to return one particular row, we only need to query based on the desired primary key value. While primary keys make data retrieval within a certain *table* very efficient, Oracle stores another type of key value that can uniquely identify a single row in an entire *database*. In Oracle, every row of every table contains a value called the **ROWID**. The purpose of the ROWID is to uniquely identify a row globally—throughout the database. Thus, no two rows in the database will ever contain the same ROWID. In addition, the ROWID is a pointer to the *actual physical location of a row* in the database. The ROWID can not only globally identify the row, but also point to its exact physical location—all the way to the blocks of data stored within a data file. ROWIDs are stored in each table as a **pseudocolumn**—it is present in the table, but not readily visible. We can, however, see the ROWID if we know where to look. The following screenshot contains a query we can use to retrieve the ROWID and `employee_id` column values in the `employee` table:



Notice that in previous queries where we used `SELECT *` to query all columns in a table, the ROWID was not present. The ROWID can only be queried directly. We must specify it by name to see it. While the ROWID might look indecipherable to us, to Oracle it contains all the needed information to uniquely identify a row's location on disk. The results you see in your query will differ from those in the output shown here.

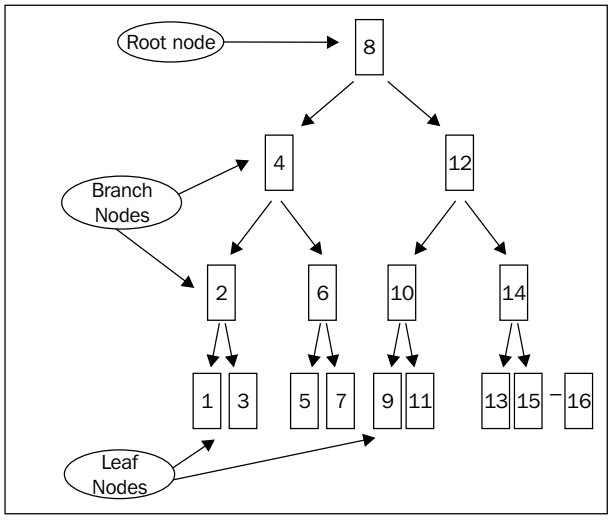
What makes a ROWID useful is that it represents the fastest access path to any given row in the database. If we could leverage ROWIDs in our queries, they would perform much faster than simple table scans. Unfortunately, while a column value such as "Johnson" for last name has meaning to us, a ROWID value such as "`AAhoSaAN+AAAAGNAAA`" does not, making it difficult to use. Fortunately, Oracle provides a structure that *can* leverage them.

SQL in the real world

While it may be tempting to think that directly using a ROWID for every query is a good idea, Oracle recommends that you do not. The physical location of rows in the database can change with certain operations and, since the ROWID represents the physical location of the row, it can change as well. This makes it unreliable as a key value. However, in some large database environments, such as data warehouses, the ROWID can be used as a key value for rows with read-only data.

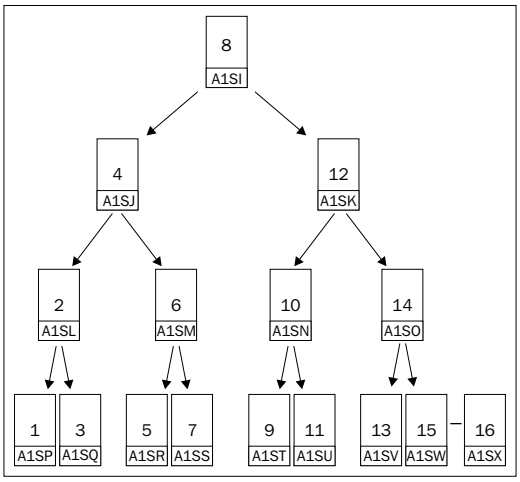
Examining B-tree indexes

To speed the access path to data, Oracle can make use of an index. An **index** is a structure that takes column and ROWID values and structures them in such a way that the rows can be accessed by their direct physical location on disk. Typical tables in Oracle are called **heap-organized tables**—that is, the table's rows are stored in no particular order. Thus, there is no guarantee that a query of the `employee_id` column will retrieve a value of 100 any *sooner* than a value of 1,000. An index, on the other hand, is stored in a particular structure that facilitates the quick retrieval of data. The most common structure for an index is the **B-tree** or **balanced tree** structure (sometimes referred to as a *binary tree*). A B-tree is a type of tree-shaped data structure used to organize data. In a B-tree, data values are read and then organized in a tree structure made up of root, branch, and leaf nodes. The following diagram gives a simple example of a B-tree structure that organizes 16 values, much like the 16 `employee_id` values of our `employee` table.



Finding any given value structured within a B-tree is a matter of starting at the root node and making *greater than, less than, equal to* choices down the tree. For instance, if we are searching for the value **9**, we start at the root and work down. The first choice could be phrased, *Is 9 greater than, less than, or equal to 8?* 9 is greater than 8, so we move down the right side of the tree. Already, we have excluded half the possible values that could be searched. The next branch node on the tree is 12. *Is 9 greater than, less than or equal to 12?*. 9 is less than 12, so we move down the left side of the remaining tree. Our next branch node is 10. *Is 9 greater than, less than or equal to 10?* 9 is less than 10, so we move down again to the left, landing at 9. *Is 9 greater than, less than or equal to 9?* 9 is equal to 9, so we have found our value.

This is fine for single values, but how can a B-tree structure help us find a particular row in the database? A B-tree index stores two values—the value that is indexed and the ROWID. Thus, once you have found the key value in the B-tree, you have also found the ROWID of the particular row, giving you the physical location of the row on disk. Thus, we could alter our diagram to include simulated ROWIDs to see how this works, as follows:



We could follow the same previous example and search through the tree for the value 9. Once we have found it, we have also found the ROWID (A1ST, in this case) for the row that locates all the values for `employee_id #9`. The example we've used is greatly simplified, but it does show the way a B-tree index is structured and demonstrates why an index can speed data access. A true index contains its data in the leaf blocks and only uses the branches for decision making. In our example, instead of doing a full table scan through 16 rows, we're only required to make four choices to arrive at the desired ROWID.

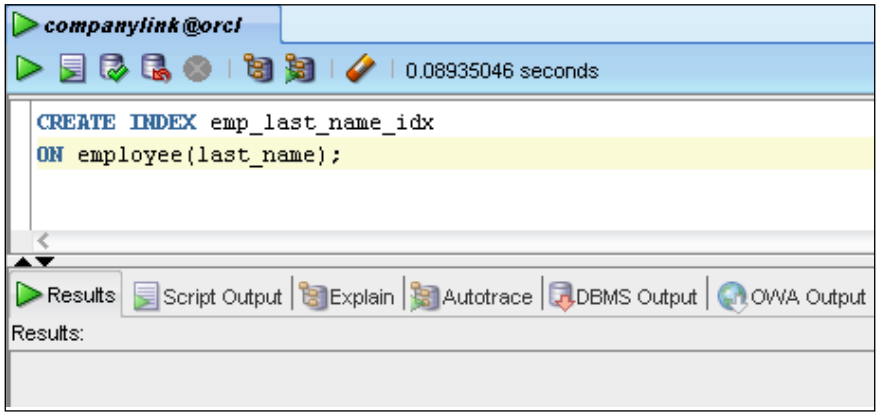
Creating B-tree indexes

An index is created using the `CREATE INDEX` command shown as follows:

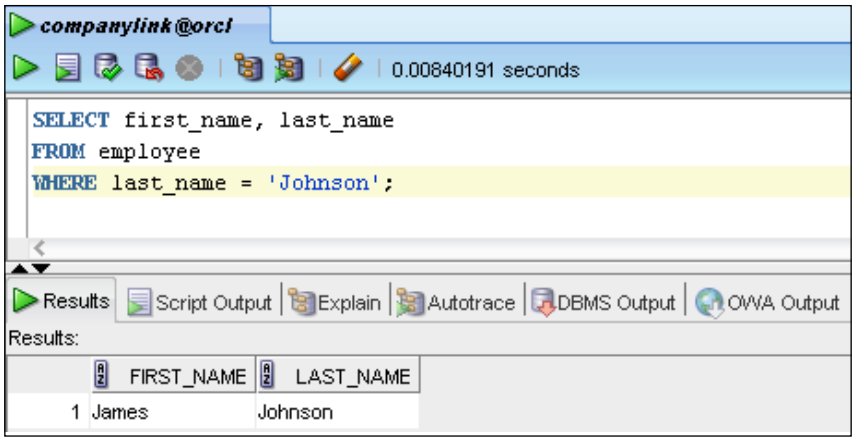
```
CREATE INDEX <index_name>
ON <table_name> (<column_1, column_2, column_x);
```

The `CREATE INDEX` command bears some resemblance to other similar commands that create database objects. It begins with a `CREATE INDEX` clause followed by the name of the index. Next, the `ON` clause precedes the name of the table and columns to be indexed, with the column or columns in parentheses. Other clauses, such as those that specify storage details, are optional. For our purposes, we want to understand how indexes work and ways they can improve our SQL.

For example, say that we've determined that our application will frequently do queries against the `employee` table whose limiting condition is often based on the employee's last name. In order to retrieve those rows more efficiently using an indexed scan, we place an index on the `last_name` column of the `employee` table. We do this using the `CREATE INDEX` command, as shown in the following example:



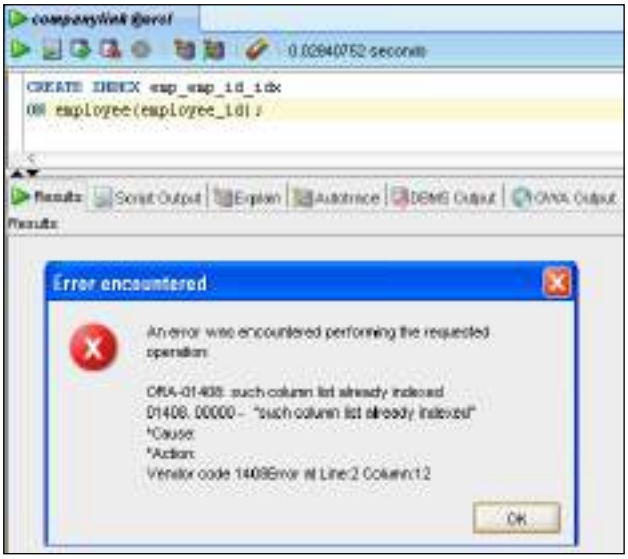
We name the index `emp_last_name_idx` and specify `employee` as the table and `last_name` as the column to index. When the `CREATE INDEX` statement is run, Oracle reads all of the values for `last_name` in the table and organizes them into a B-tree structure, along with their respective ROWID values. We should note two facts from this. First, the index is persistently stored in the database as an object, much like a table. It can be used as long as it exists. Second, since the index stores values, it takes up physical space in the database, although comparatively little to that of its table. The next time we issue a query such as the one shown in the following screenshot, the index can be used:



SQL in the real world

Notice that we said the index *can* be used. The Oracle RDBMS uses a feature called the *optimizer* to make decisions as to the best way to retrieve data. Often, this involves using an index. However, circumstances may dictate that the optimizer choose another access path, sometimes even ignoring the index. Even so, indexing the columns that are commonly used in a `WHERE` clause is a good standard practice.

It may be unclear why we chose to index the `last_name` column, since we have continually used `employee_id` in our examples. Let's try indexing the `employee_id` column in the `employee` table and see what happens.



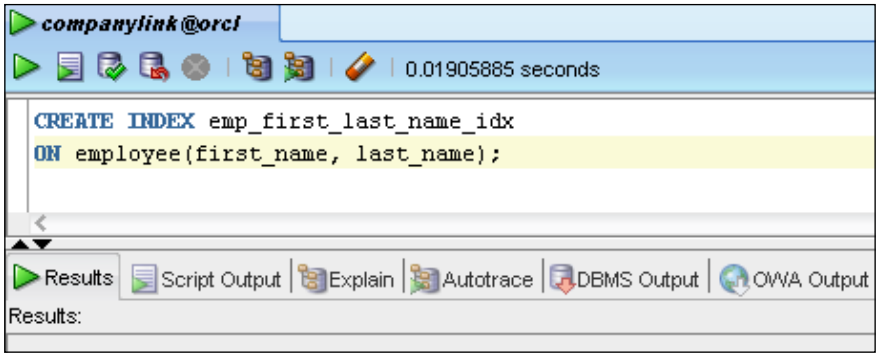
As we can see, we receive an error saying, **"such column list already indexed"**. But, how could this be? At this point, we have only created an index for the `last_name` column in the `employee` table, not `employee_id`. Actually, we created the index on `employee_id` in the last chapter without even recognizing it. This is because we created a primary key on the `employee_id` column. Any time a primary key is created, an index is automatically created for the column. Since primary key columns are commonly queried, this is another reason to establish primary keys on relational tables. We also have the option of creating the index first, then instructing Oracle to use that index during the creation of the primary key.

Using composite B-tree indexes

In the same way that we can create composite primary and foreign keys, we can also create **composite indexes**—indexes on more than one column. Say, for example, that we frequently issue a query that uses the following `WHERE` clause:

```
WHERE first_name = <some first name>
AND last_name = <some last name>;
```

Such a query most likely cannot use our index on `last_name` to much benefit, since the `first_name` column is not indexed with it. If we frequently query columns with multiple conditions using operators such as `AND`, we need to create a composite index, as shown in the following screenshot:



Here, we index two columns, `first_name` and `last_name`, in that order. Now our `WHERE` clause with multiple conditions can make use of our composite index for faster data access. It is very important to notice that even though we previously indexed the `last_name` column by itself, *we can still index it here* without receiving an error. Because they index by different values, indexing `last_name` alone is different than indexing it with another column. This would be true even if we reversed the order of the composite index, using `last_name`, then `first_name`. The key values are different, so another index can be created.

SQL in the real world

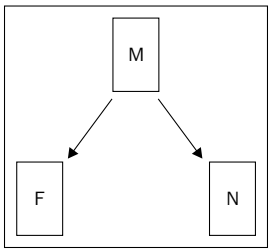
Index names can sometimes seem long and confusing. Although index naming is normally dictated by coding standards, a common way to denote index names is to abbreviate the table and column names, then suffix with `_idx` or `_ind`. Using this standard, we can see that an index called `emp_emp_id_idx` is an index on the `employee` table (`emp`) on `employee_id` (`emp_id`). We put both together with an `_idx` suffix to form the name. Decoding index names is just a matter of knowing how they're constructed.

Working with bitmap indexes

B-tree indexes are by far the most common type of index used with Oracle. However, a B-tree index isn't always the most optimal type available. In fact, the type of data being indexed can often render a B-tree index useless. This section examines just such a situation.

Understanding the concept of cardinality

Consider for a moment indexing a different column in the `employee` table—gender. If we assume that the gender column has three possible values, M, F, and N, we could display a B-tree structure for the index on this column as shown in the following diagram:

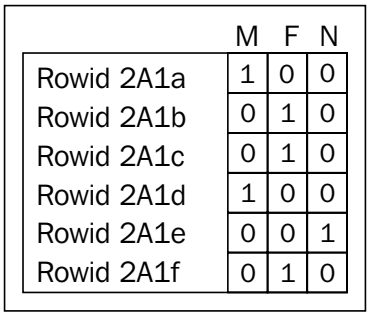


As we can see, the structure itself is very brief. Since there are only three possible values, there are only three nodes possible in the index. Each ROWID falls under one of those nodes. However, the limited number of nodes in the index has affected the efficiency of the index. The strength of a B-tree index lies in its ability to quickly eliminate the need to scan a large percentage of the structure using a few quick decisions. Our index on gender leaves us with a large number of ROWIDs under each node that must now be scanned, so we've effectively lost the benefit of our index. The problems with using B-tree indexes on columns that have only a few distinct values are due to their cardinality and selectivity.

In set theory, **cardinality** refers to the number of distinct or unique values in a set. We can also apply this term to refer to the set of values in a column. For our purposes, a column with many distinct values, such as our primary key on `employee_id`, is said to have a *high cardinality*. Conversely, a column with a limited number of distinct values, such as our gender column, has a *low cardinality*. From cardinality, we can also determine selectivity. **Selectivity** is calculated by taking the total number of rows in a table and dividing it by the cardinality. Thus, if our `employee` table has 1,000 rows and the gender column has three distinct possible values, our selectivity is 1000/3, or approximately 333, which is considered very high. The lowest possible selectivity would be found on, for example, a primary key column. Since a primary key can have no duplicate values, the selectivity for any given column would always yield one. If the table held 1,000 rows, we know that they are all distinct, giving us a cardinality of 1,000. Dividing 1,000 rows by a cardinality of 1,000 would yield a selectivity of one. The same would be true if the table had 10,000 or a million rows. They are all distinct, so the selectivity is one.

Examining the structure of bitmap indexes

Armed with these terms, we could combine this with what we've seen about B-tree indexes and make a generalization: *B-tree indexes are most effective on columns with a low selectivity*. We could also say that they are highly *ineffective* on columns with high selectivity. It is this ineffectiveness that has led Oracle to offer a different kind of index. A **bitmap index** uses a different type of structure than the B-tree. Rather than using a tree structure, a bitmap index organizes key values and ROWIDs in a bitmap, as shown in the following diagram:



	M	F	N
Rowid 2A1a	1	0	0
Rowid 2A1b	0	1	0
Rowid 2A1c	0	1	0
Rowid 2A1d	1	0	0
Rowid 2A1e	0	0	1
Rowid 2A1f	0	1	0

Here, each distinct value for the gender column, M, F, and N, is listed along the top. Below these values is the bitmap—a structure of ones and zeros for each ROWID in the table. Each *row* in the bitmap will have a single value marked with a *positive bit* while the others will have zeros. The example indicates that the first and fourth rows in the bitmap index are positive for the value 'M' for male. The second, third, and sixth rows have a positive bit under value 'F' for female. The fifth has a positive bit for value 'N', indicating that the fifth row in the table contains 'N' in the gender column.

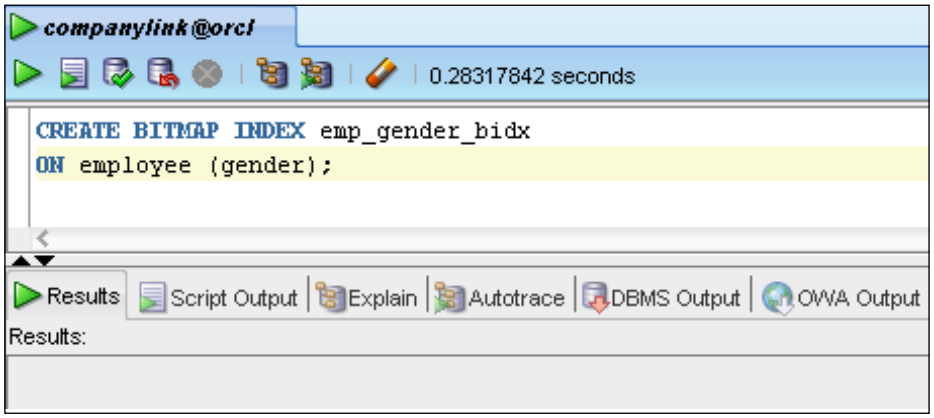
The benefit of a bitmap index is that it provides us with a way to quickly retrieve ROWIDs from high selectivity columns. If a bitmap index, such as the one shown previously, was in place on the gender column, it could be used for the following query:

```
SELECT * FROM employee
WHERE gender = 'M';
```

When this query is executed, the index is scanned. Because it is structured as a bitmap, the scan only has to search the bitmap for positive bits for 'M'. From that scan, the ROWIDs are retrieved and the data is presented.

Creating a bitmap index

Bitmap indexes are created similar to B-tree indexes, requiring only the addition of the BITMAP clause, as shown in the following screenshot:

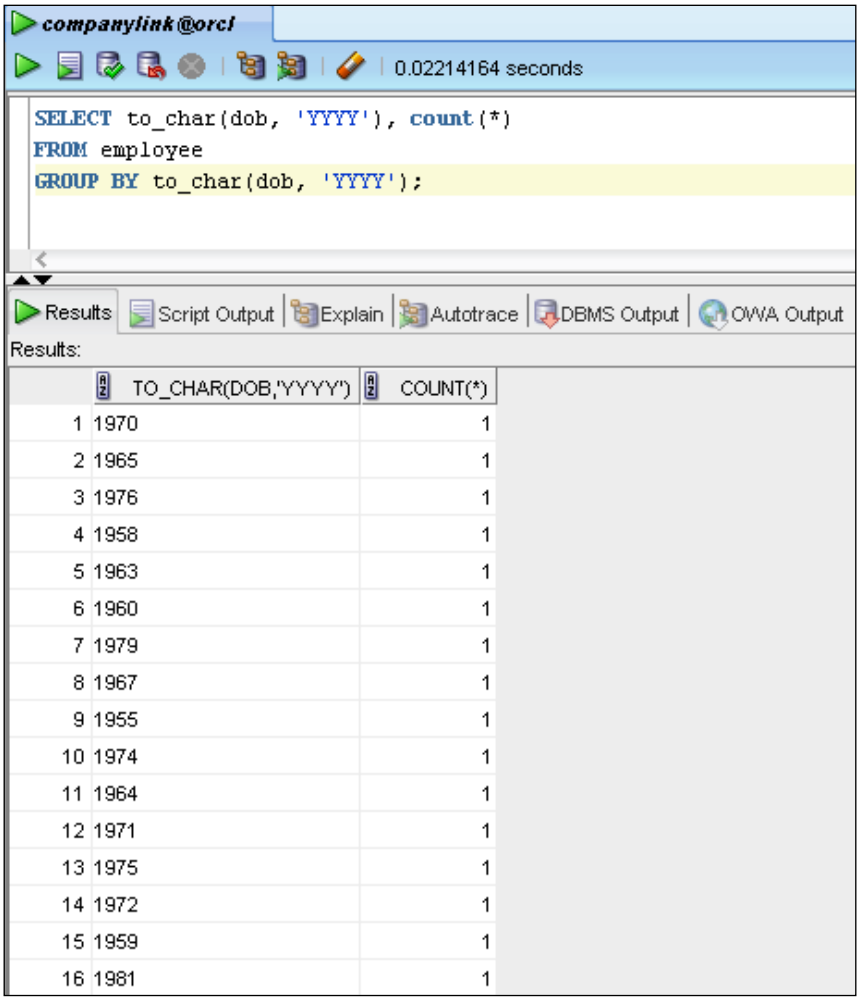


As we've previously seen, we specify a name, in this case, `emp_gender_bidx`, and the table and column to be indexed. The values for gender are read and constructed into a bitmap matching those characteristics. Now, any future queries on the highly selective column, gender, can make use of an effective index.

Working with function-based indexes

In earlier chapters, we made extensive use of both single and multi-row functions. Using indexes in conjunction with functions often presents certain challenges. To understand why, it is important to remember once again how indexes are stored. Recall that indexes store the indexed values along with ROWIDs—nothing else.

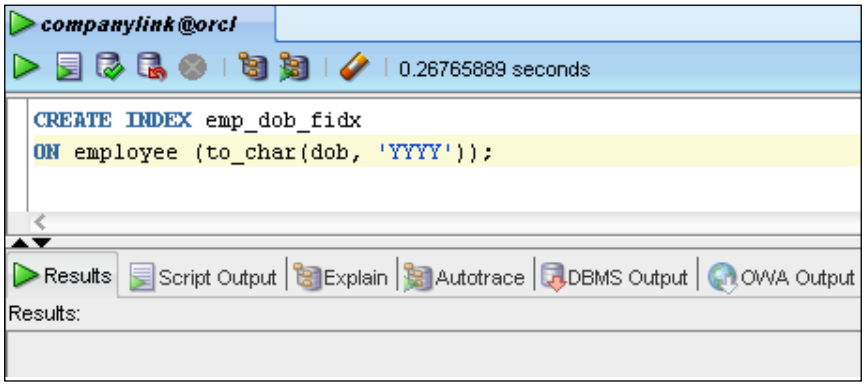
Consider the following example. Say that we want to do a count of employees grouped by the year of their birth. We've seen queries such as this that use a `GROUP BY` clause and extract the year using the `TO_CHAR` function, as shown in the following example:



	TO_CHAR(DOB,'YYYY')	COUNT(*)
1	1970	1
2	1965	1
3	1976	1
4	1958	1
5	1963	1
6	1960	1
7	1979	1
8	1967	1
9	1955	1
10	1974	1
11	1964	1
12	1971	1
13	1975	1
14	1972	1
15	1959	1
16	1981	1

Certainly this query would execute normally, but would it execute optimally? If the table was very large, we would need an index on the DOB column to receive the highest return on performance. However, it turns out that neither a B-tree nor a bitmap index on the DOB column would give us any performance benefit for the query in the previous screenshot. Why would this be the case? Creating an index on the DOB column will index each of the values in the column along with their respective ROWIDs. However, we are querying the DOB column in a *way that is different than the way they are indexed*. We are not querying the DOB column itself, but rather a subset of each value—the value for four-digit year. Thus, a B-tree or bitmap index would be useless, since the values for year have not been explicitly indexed. In fact, Oracle would not even attempt to use either type of index in such a query. We need a new kind of index.

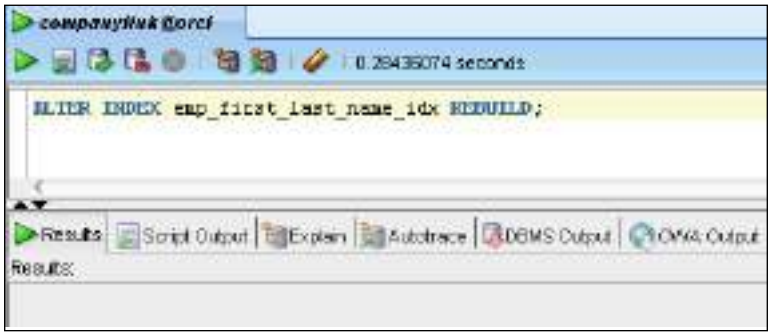
For situations such as the previous one, Oracle provides **function-based indexes**. Function-based indexes allow you to index values based on a function instead of typical column values. An example is shown in the following screenshot:



As we can see, creating a function-based index does not require any new clauses. We simply specify a function instead of only a column name. After this index is created, Oracle has the opportunity to make use of it whenever we query based on that particular function. However, remember that the function used in the query must match the one used in the index. It does no good to index a column based on a `TO_CHAR()` function if the query uses a `SUBSTR()`.

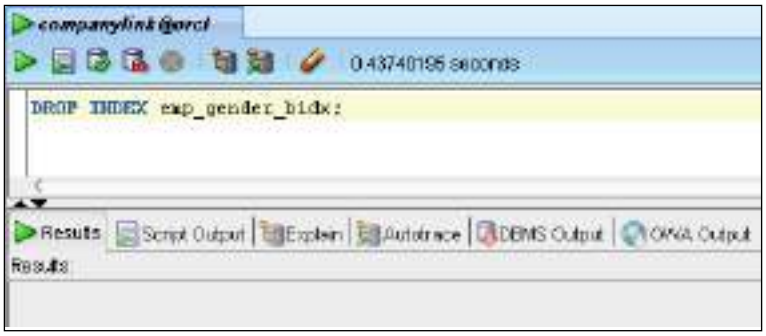
Modifying and dropping indexes

The structure of an index is built during its creation. However, the data in a table changes over time. Data is added, changed, and removed using DDL statements. Every time a table is changed, the index must be changed to reflect that. Highly dynamic tables can lead to index *fragmentation*, particularly when deletes are performed. For these reasons, it is often necessary to *rebuild* an index. When an index is rebuilt, the table is re-scanned and the index is restructured accordingly. We use the `ALTER INDEX` command to rebuild an index, as shown in the following example:



The `ALTER INDEX... REBUILD` syntax works for each of the index types that we have discussed. Thus, it is not necessary to use the `BITMAP` keyword when rebuilding a bitmap index. Using `ALTER INDEX` to rebuild an index is beneficial in that it can be done while the table and index are online. You do not need to drop and recreate the index. `ALTER INDEX` can also be used to perform other operations besides index rebuilds, such as those dealing with space management.

When we need to completely remove an index, we use the `DROP INDEX` command, as you might expect by now. Like `ALTER INDEX`, the `DROP INDEX` syntax works for each of the types of indexes we have seen, without any qualifying clauses such as `BITMAP`. Dropping an index is shown in the next statement:

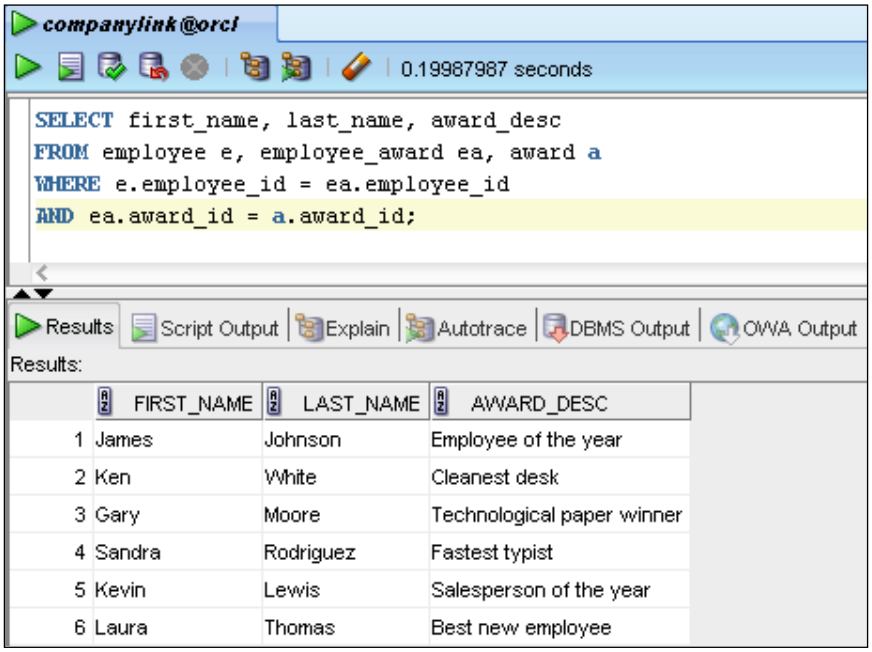


Working with views

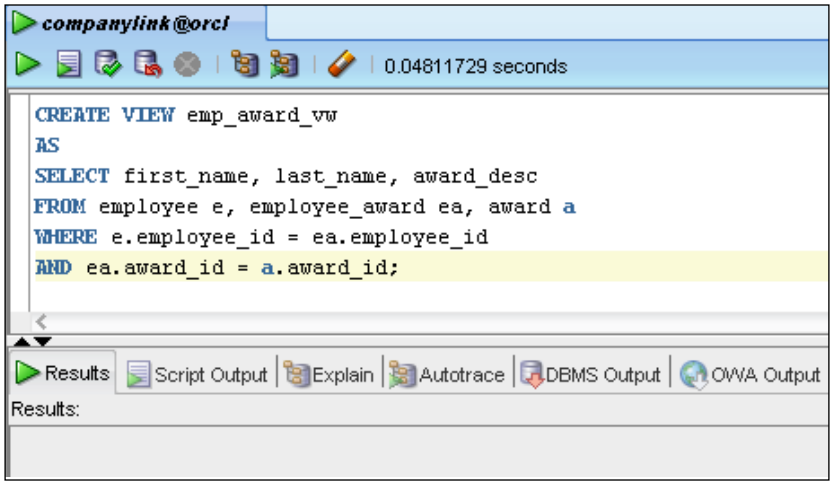
Although we began our study of SQL by using some very simple queries, at this point we have definitely graduated to some fairly complex ones. Multi-table joins, subqueries, and nested functions can all play a part in the need for long and complicated SQL code. Often, this code is created to be reused numerous times for business needs such as reporting. A **view** allows us to store a query as a database object so that it is simplified and reusable.

Creating a view

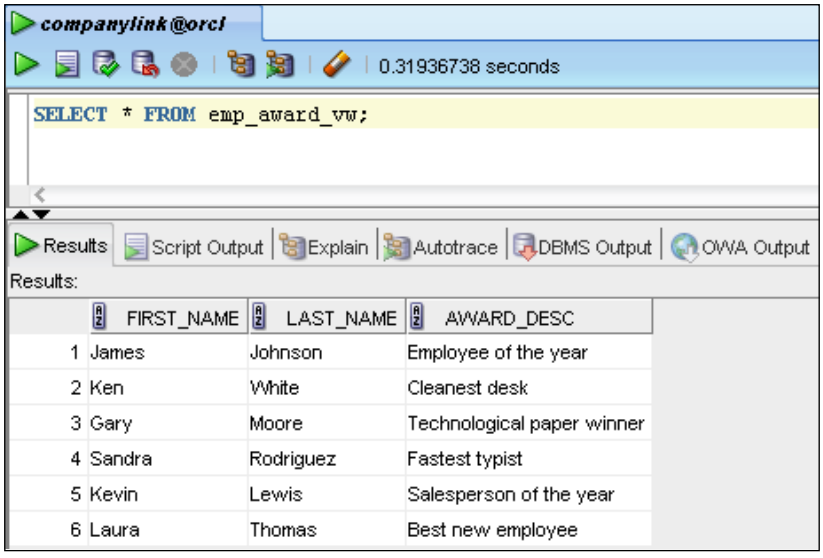
Since views are objects that store queries, we should look at an example of the type of `SELECT` statement that can be stored as a view. The following example contains a SQL statement we used in *Chapter 5*, *Combining Data from Multiple Tables* to create a reasonably complex multi-table join of `employee`, `employee_award`, and `award`. This is our candidate statement to simplify using a view.



Database views are creating using the `CREATE VIEW` command, as shown in the following example. We use a view to encapsulate the previous `SELECT` statement into a view.



We construct the statement syntactically using `CREATE VIEW`, a name for the view, and the keyword `AS`. Following this part of the statement, we simply add the code for the query we want the view to represent. Now, we can access the data returned from the complex SQL statement by selecting from the view in the same way we would select from a table, as shown here:



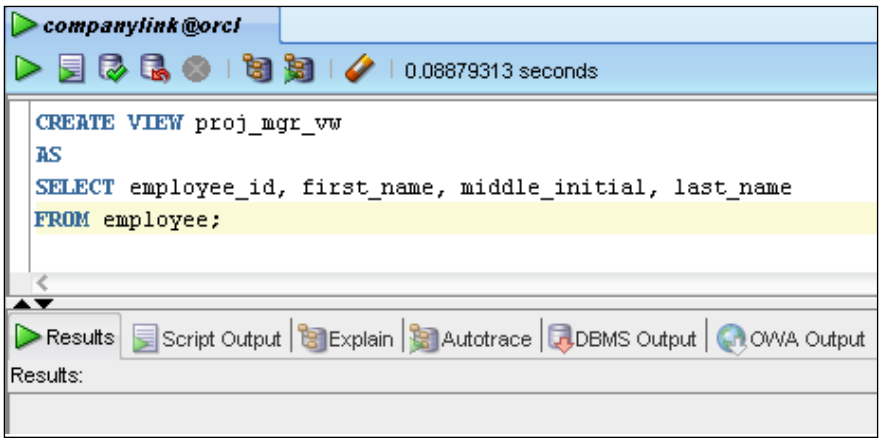
	FIRST_NAME	LAST_NAME	AWARD_DESC
1	James	Johnson	Employee of the year
2	Ken	White	Cleanest desk
3	Gary	Moore	Technological paper winner
4	Sandra	Rodriguez	Fastest typist
5	Kevin	Lewis	Salesperson of the year
6	Laura	Thomas	Best new employee

Selecting from the view returns the identical rows that would have been returned if we had run the original statement. We have essentially used a view to reduce a complex query into a simple one. One important fact to take away from this is that *views contain no storage of their own*. The storage required for a view is only the amount needed to store the query. Thus, views take up essentially zero storage space in the database.

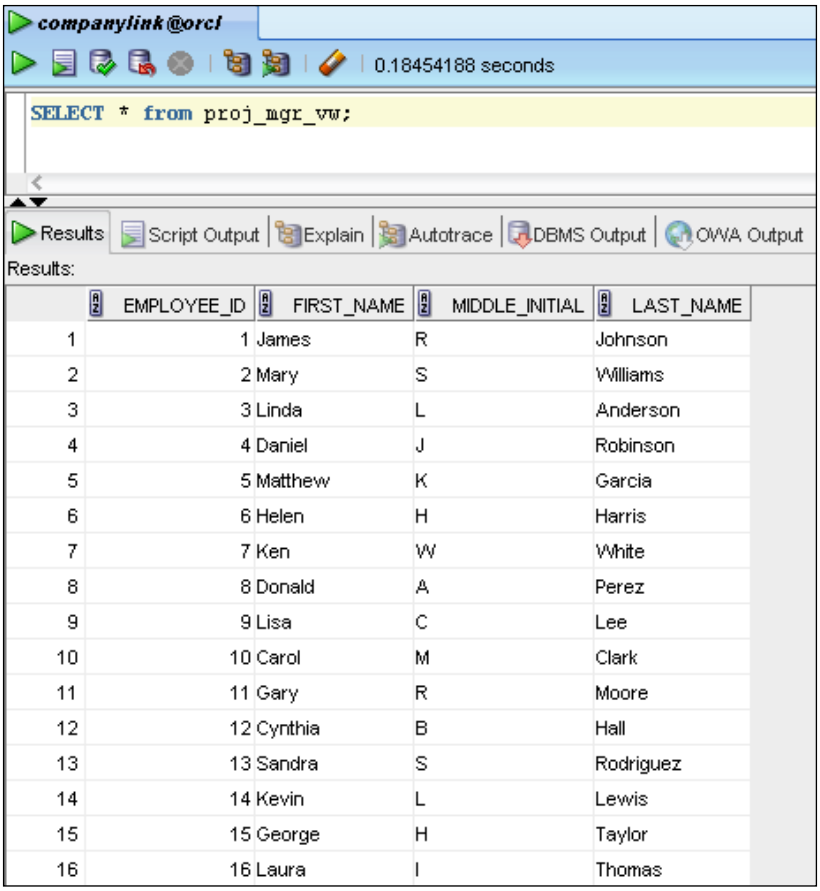
With certain configurations of Oracle, the `CREATE VIEW` statement shown previously can generate an error regarding a product called Trusted Oracle—a special security package. If so, do not be concerned. The code shown is the correct syntax to create the view, although it requires a workaround in rare cases.

Creating selective views

Another way views are sometimes used is for security. Although not a strict type of security, views can be used for **data hiding**. Let's take an example from our `Companylink` database. Say that we want to allow project managers to have visibility into certain columns in the `employee` table, but not others. To hide these columns, we could give project managers access to the data using a view, instead of directly into the table. We create such views by defining the columns that we wish to allow, as shown in the next example:



Using a column selective view, the project managers have access to only that data they require. Now, when they wish to see employee information, they access it using the view, as shown in the following screenshot. Again, this prevents the need for maintaining a second limited copy of the table just for project managers.

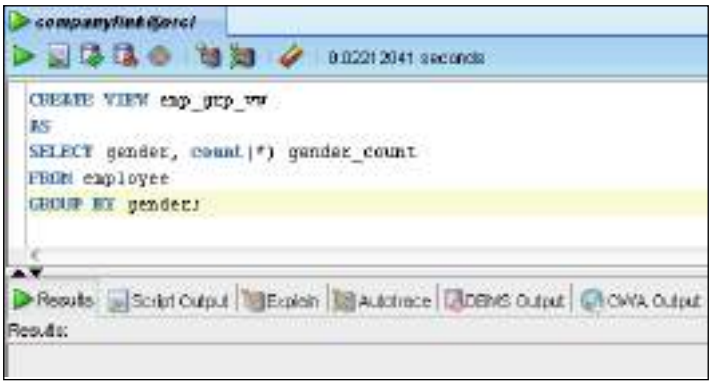


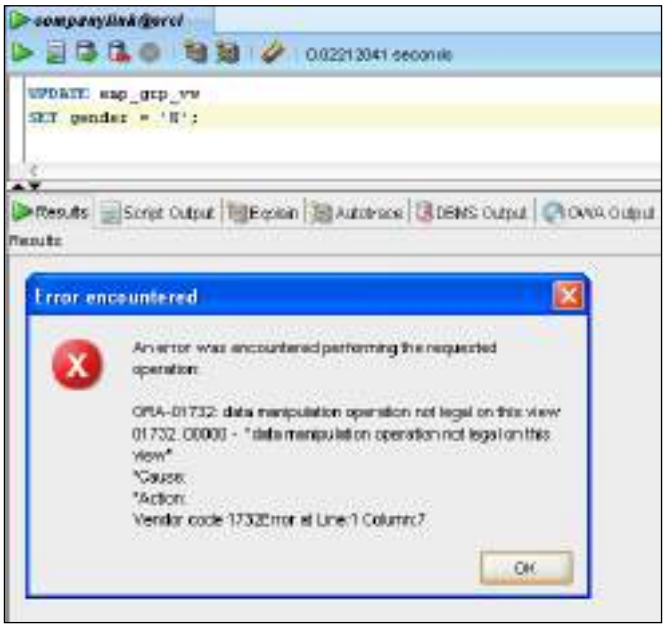
	EMPLOYEE_ID	FIRST_NAME	MIDDLE_INITIAL	LAST_NAME
1	1	James	R	Johnson
2	2	Mary	S	Williams
3	3	Linda	L	Anderson
4	4	Daniel	J	Robinson
5	5	Matthew	K	Garcia
6	6	Helen	H	Harris
7	7	Ken	W	White
8	8	Donald	A	Perez
9	9	Lisa	C	Lee
10	10	Carol	M	Clark
11	11	Gary	R	Moore
12	12	Cynthia	B	Hall
13	13	Sandra	S	Rodriguez
14	14	Kevin	L	Lewis
15	15	George	H	Taylor
16	16	Laura	I	Thomas

Distinguishing simple and complex views

We could say that views are a *window* into one or more tables. This window includes the ability to execute DML statements such as `INSERT`, `UPDATE`, and `DELETE` through a view, but only with certain restrictions. An **updatable view** is one that allows you to manipulate the underlying table data through DML. To understand the restrictions on updatable views, it is important to make a distinction between simple views and complex views.

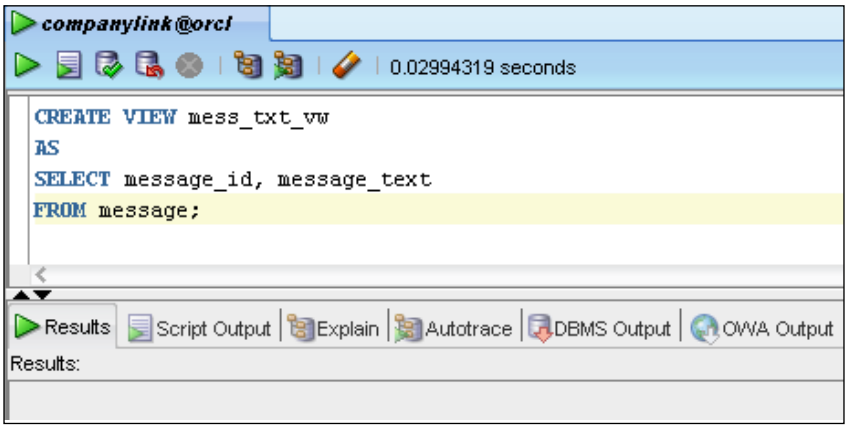
In Oracle, a **simple view** is one that selects data from a single table, contains no functions, and performs no aggregation operations. With certain limitations, we can perform DML operations on a simple view. Conversely, a **complex view** is one that can draw data from multiple tables, utilize functions, and can perform aggregation operations such as grouping data. It is generally very difficult to execute DML operations on complex views, although not impossible. The problem lies with the inability to map the rows seen by the views back to the base table. For instance, say we create a view that uses a `GROUP BY` statement. If we attempt to update a row *in the view* then there is no way to identify exactly which row in the base table is to be updated, since the data is grouped from the frame of reference of the view. We see this in the following two illustrations:

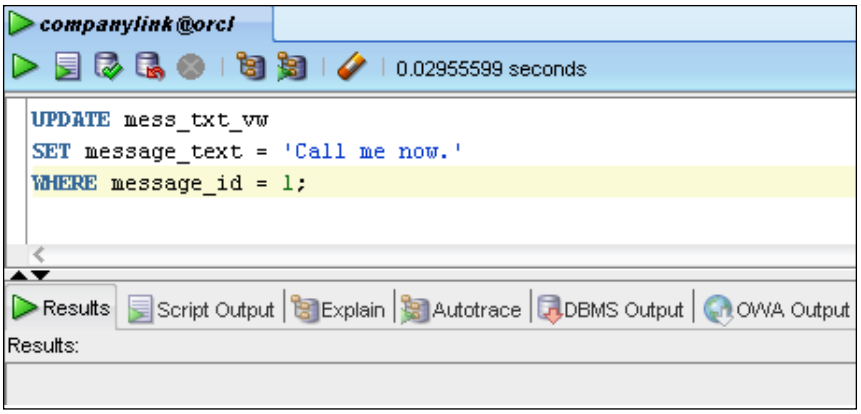




First, notice that we qualified our `COUNT()` function in the view with a column alias. In a view, every column must have a unique name. So, in our example, we must alias any functions we use such as `SUM()`, `AVG()`, or `COUNT()` with an alias. However, when we attempt to update the view, we receive an error stating, *data manipulation operation not legal on this view*. This is consistent with what we know, owing to the fact that Oracle cannot find a one-to-one mapping of the data in the view to the data in the underlying table.

We can, however, manipulate rows in a simple view, with one very important caveat. In general, DML can only be done against a view if the table is key-preserved. A **key-preserved table** is one in which every unique key value is also unique in the view. We can see this behavior in the following two examples:





Configuring other view options

Views can be created with optional keywords that allow greater control. The syntax tree for views with optional keywords is listed as follows:

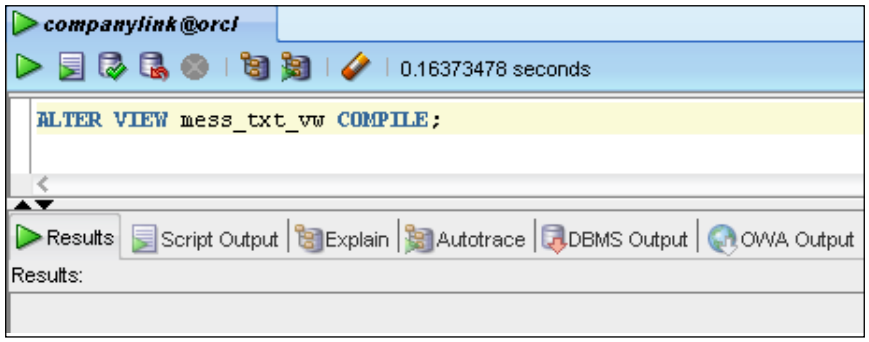
```
CREATE [OR REPLACE] [FORCE] VIEW {view_name}
AS
{query}
[WITH CHECK OPTION];
```

The options are defined as follows:

- `OR REPLACE` – The existing view will be dropped and re-created in one step
- `FORCE` – The view will be created even if the underlying table in the `SELECT` statement does not exist
- `WITH CHECK OPTION` – This will prevent changes from being made to the underlying table that would cause rows to disappear from the view

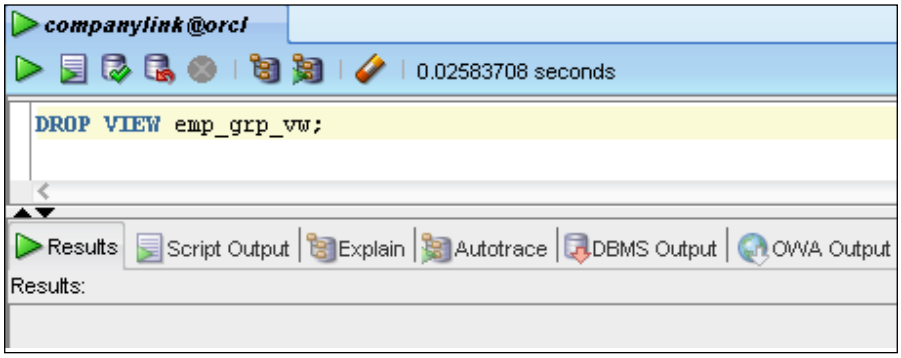
Changing or removing a view

A database view can be re-structured using the `ALTER VIEW` command. `ALTER VIEW` is generally used to recompile the view. During recompilation, the view is checked to ensure that all columns and tables in the underlying query still exist. The use of `ALTER VIEW` is shown in the following example:



In the event that any of the columns or tables had been dropped or their names had been changed, the recompilation would reveal this.

To drop a view, we use the `DROP VIEW` command. This will remove the view from the database. In the following example, we remove a complex view created earlier in the chapter:



Using sequences

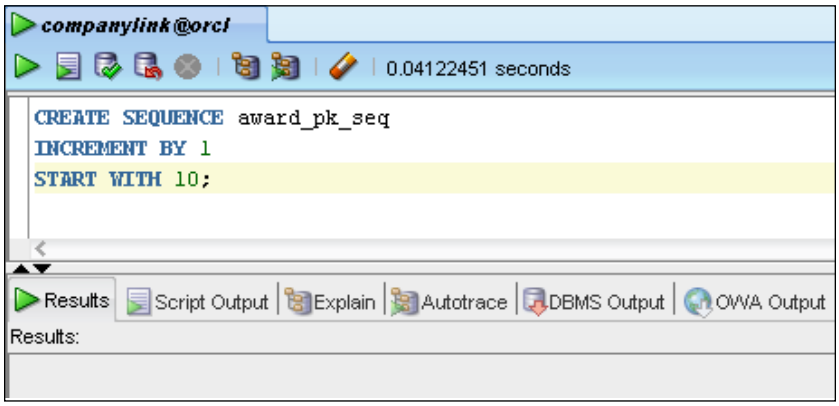
In the previous chapter, we discussed two different methods for handling primary keys. Natural keys are key values that have actual meaning in the business model. They could include social security numbers or account numbers. However, our *Companylink* model uses **synthetic keys**—key values that are generated to maintain uniqueness. Using synthetic keys provides an easy way to guarantee uniqueness, although some object to the additional space required for non-business data. This is often a small price to pay for the usefulness of a synthetic key. Synthetic keys do, however, need to be generated. This can be done using programmatic techniques in the originating application. It can also be done using sequences. A **sequence** is a database object capable of generating sequential integers. These numbers can then be used as key values.

Using sequences to generate primary keys

Say that we want to be able to add new types of awards to the `award` table. In our last chapter, we placed a primary key on the `award_id` column, so we need to guarantee that only unique values are contained therein. One method to do this would be programmatic. A routine could be written that would look up the largest value in the table, increment it by one, and then insert the value. Although this method could work, there are at least two problems with it. First, if the table is heavily used, the rows would need to be locked to prevent another user session from doing the same thing a split-second later. It is possible that session #1 could query for the maximum value and that session #2 could do the same a moment after, but before session #1 inserts the value. In this scenario, both sessions increment the same

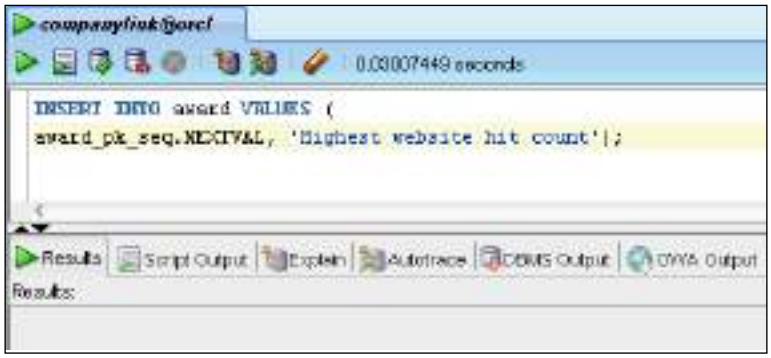
value, but only one of them can insert it without causing a primary key violation. The second problem with the programmatic approach is performance. In order to use this method, many extra table scans must be performed to find the maximum value to increment. However, we can avoid both of these problems through the use of sequences.

The ability to generate unique values for a synthetic key is one of the most common uses of sequences. Since a table can have no more than one primary key, the sequence is designed to be *paired* with a particular table. The sequence is then used only for that table to avoid the gaps that could occur if it was used in multiple places. We could create a sequence for the `award` table in our `Companylink` database to guarantee unique primary key values, as shown in the following screenshot:

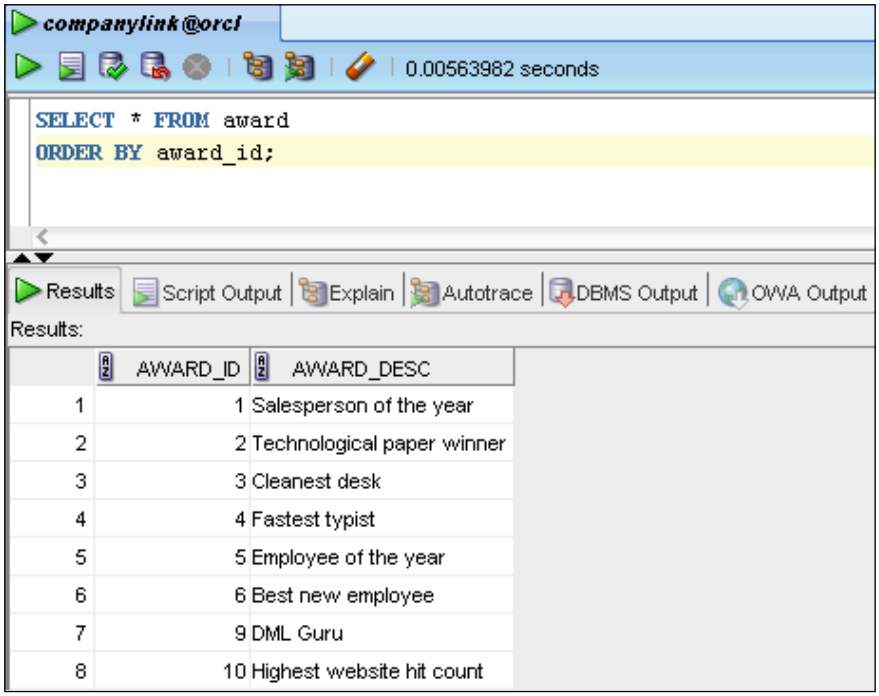


Here, we create the new sequence using a `CREATE SEQUENCE` command. The command has many options, but we look at two of them here. The first optional clause is `INCREMENT BY`, which allows us to specify the amount by which the sequence will be incremented. The default is 1, which increments the sequence by a single integer amount. If we were to specify `INCREMENT BY 10`, the sequence would be incremented by an amount of 10 each time. The second clause we've defined is `START WITH 10`. The `START WITH` clause specifies the originating integer value for the sequence. The default is `1`, but we've defined our first sequence value to be `10`, since we have existing values in the table and want to ensure that we do not violate the primary key.

When we want to make use of a sequence, we reference it by name followed by the keyword `NEXTVAL`, as demonstrated in the following screenshot:

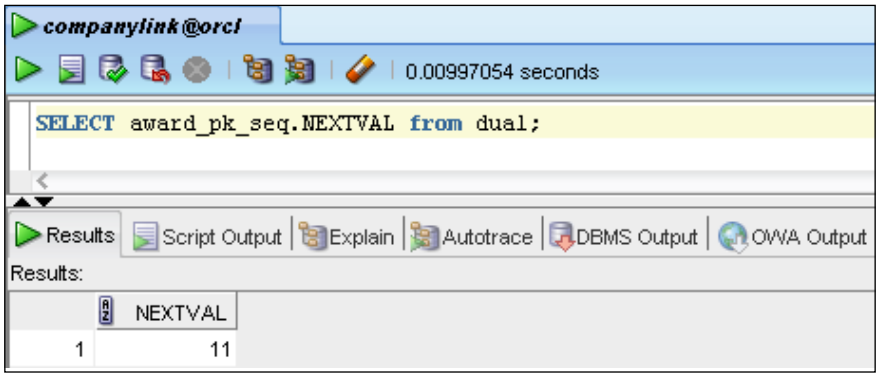


Here, we reference the sequence by name as if it were a string literal, although it is not in single quotes. Since the sequence has just been created, even though the `START WITH` value is `10`, the `NEXTVAL` will produce a `10`, essentially instantiating the sequence. We can see the results of our sequence insert by selecting the rows from the award table, as shown in this example:

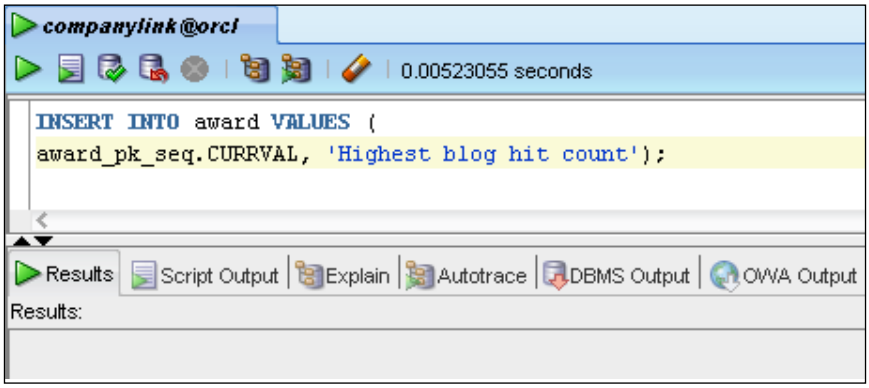


	AWARD_ID	AWARD_DESC
1	1	Salesperson of the year
2	2	Technological paper winner
3	3	Cleanest desk
4	4	Fastest typist
5	5	Employee of the year
6	6	Best new employee
7	9	DML Guru
8	10	Highest website hit count

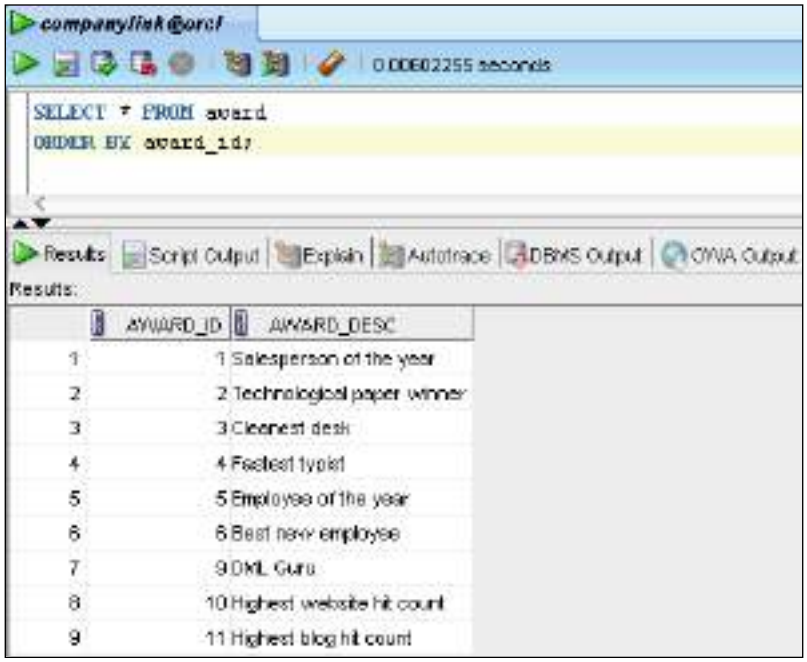
We can also manually increment the sequence using the DUAL table, as we see in the following screenshot:



Just by selecting the `NEXTVAL` from the dual pseudo-table, our sequence is incremented. Since our sequence is now increased to the number **11**, we can demonstrate the way to use `CURRVAL` to insert the current value for the sequence, as shown in the following example:



We invoke `CURRVAL` the same way as `NEXTVAL`, pairing it with the sequence name and in the same position as a literal. We see the results of our `CURRVAL` insert in the following query:



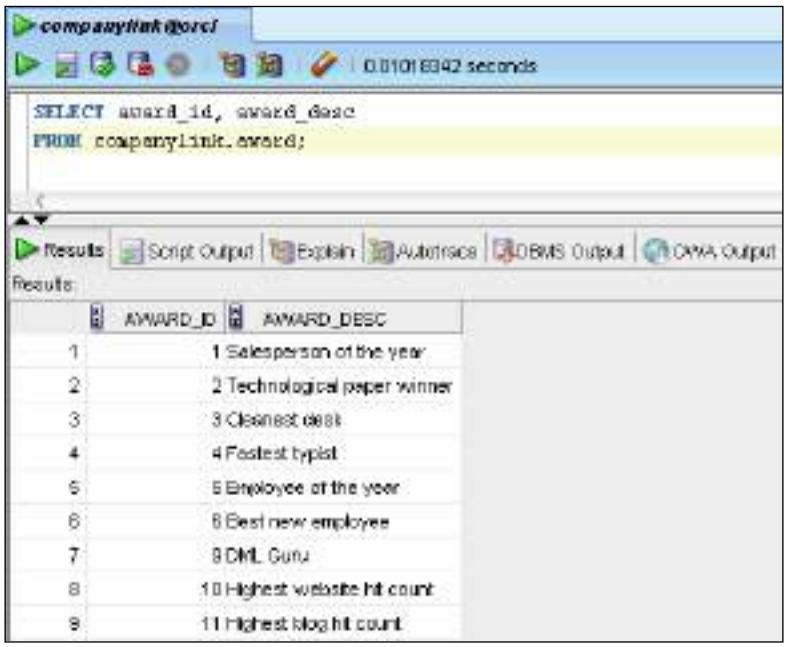
Any time a sequence is created, it must be first invoked using `NEXTVAL`. If we attempt to insert using a `CURRVAL` immediately after creating the sequence, we will receive an error, since the sequence has not been instantiated.

Object naming using synonyms

In our last chapter, we introduced the concept of a **schema**. We said that, in Oracle, no overt distinction is made between database users and object owners. When a database user creates database objects, that user owns those objects and they form the user's schema.

Schema naming

Throughout this book, we have only worked with objects that are within the *Companylink* schema. Thus, when we accessed these tables, we referred to them only by their name. Technically, however, we could reference them by their schema name as well as using dot-notation, as shown in the following example:



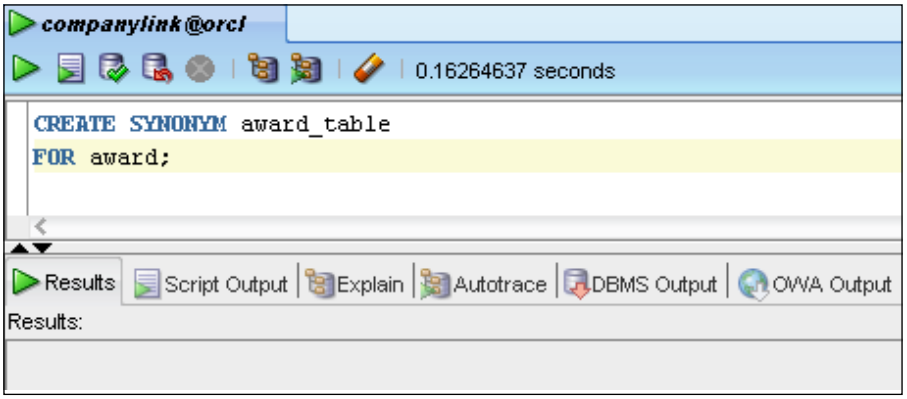
Prefixing the `award` table with `companylink` allows us refer to the table by its actual schema name. If we logged into the database with a username other than `companylink`, we would be required to do this. Thus, if we were connected to the database as the user *manager* and queried from the `award` table *without* the schema prefix, we would receive an error saying that the table doesn't exist. This is because Oracle makes the assumption that *if we do not preface the table name with its schema name, we are referencing a table in our current schema*. Therefore, in the examples throughout this book, we made no schema reference to any of the objects used, since we've always operated on objects within our *Companylink* schema.

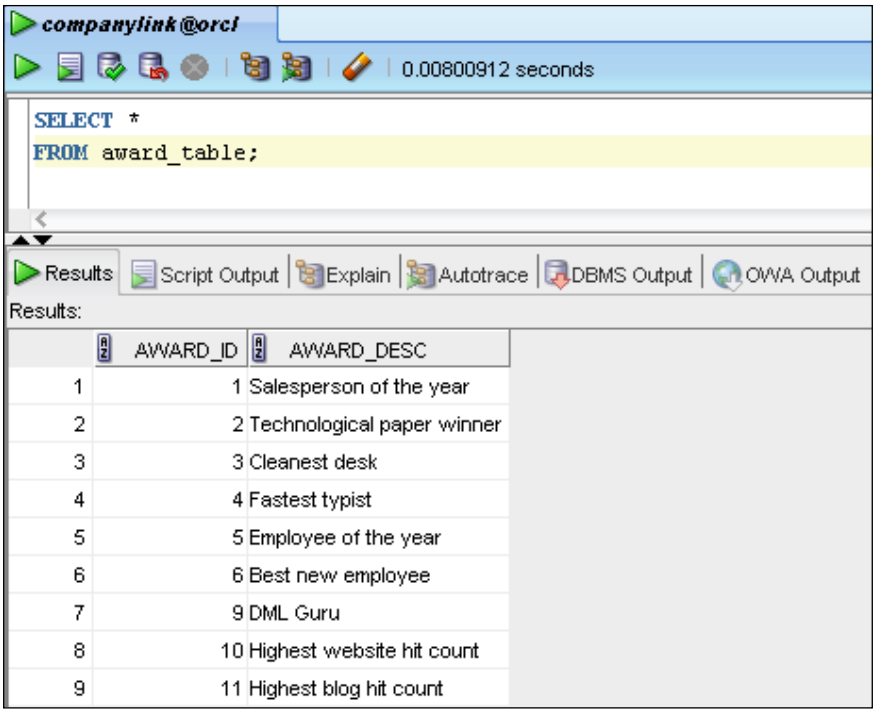
Using synonyms for alternative naming

While the dot-notation approach is a very explicit way to reference objects, it is sometimes considered inconvenient. Our examples have not required a schema reference, since all of the objects used exist within a single schema. However, in logical architectures that use multiple schemas, a user or developer would have to know the name of the object *and* the name of the schema in order to correctly reference the object. Fortunately, Oracle provides us with a type of database object, called a synonym, which allows for greater control over object naming. A **synonym** is an alternative name for any database object, although it is most commonly used for tables. Synonyms allow users to reference a table by a different name, irrespective of its true underlying name, or schema. Oracle provides us with two types of synonyms—private and public.

Creating private synonyms

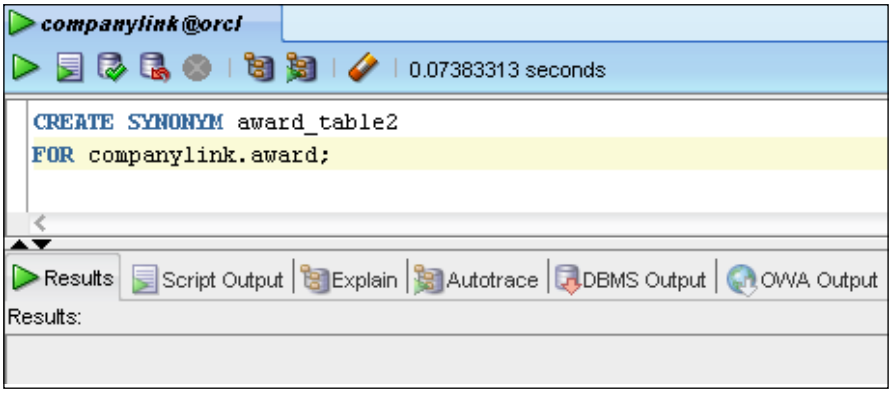
A **private synonym** is one that exists within a user's schema that can either reference an object within that schema using a different name, or refer to an object in a different schema without using its schema reference. The following two screenshots show examples of using a private synonym to create an alternate name for a table. They demonstrate the `CREATE SYNONYM` clause along with `FOR`.

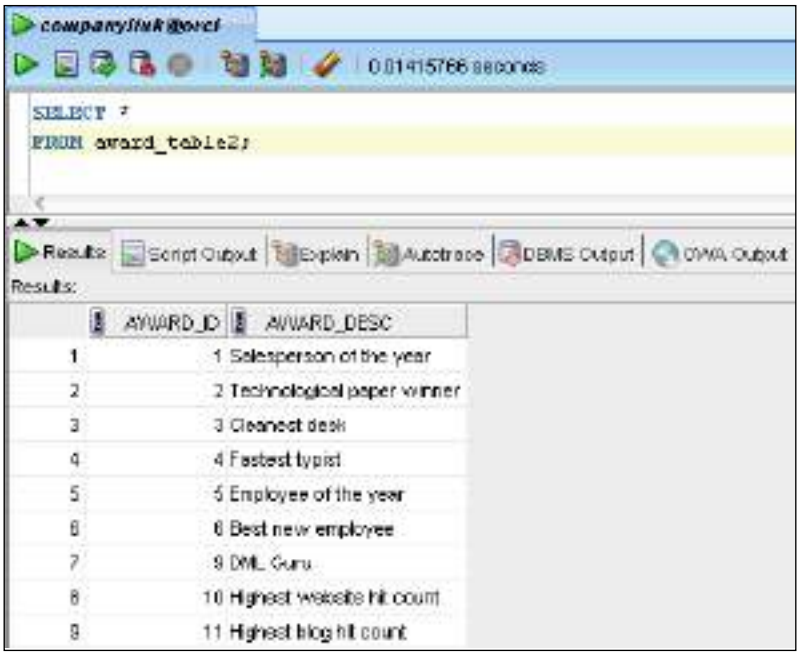




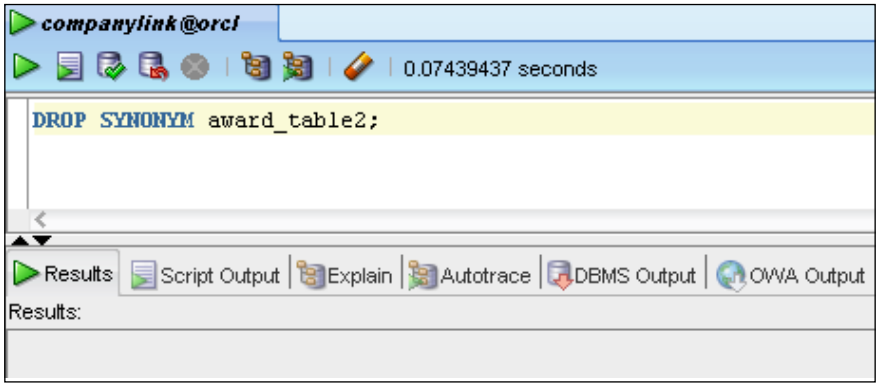
	AWARD_ID	AWARD_DESC
1	1	Salesperson of the year
2	2	Technological paper winner
3	3	Cleanest desk
4	4	Fastest typist
5	5	Employee of the year
6	6	Best new employee
7	9	DML Guru
8	10	Highest website hit count
9	11	Highest blog hit count

When we need to create a synonym that references a table in another schema, we reference the table using its schema name, as shown in the following two examples:



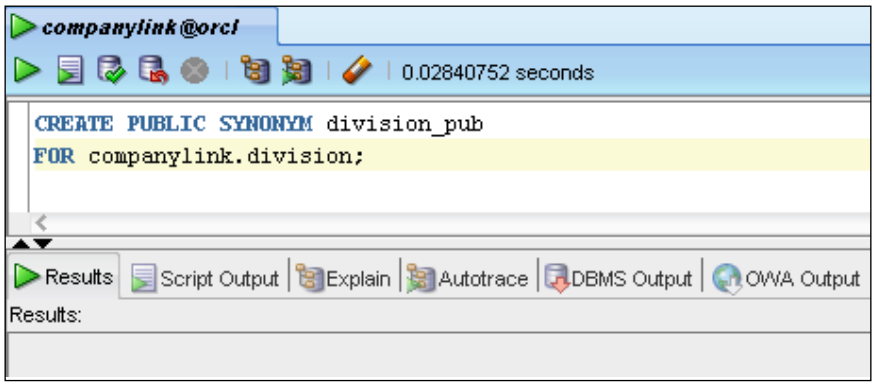


Here, even though the `award` table is in the same schema, we can reference it using dot-notation, just as we would if the table were in another schema. When we want to remove a private synonym, we use the `DROP SYNONYM` command, as shown in the following screenshot:



Creating public synonyms

The second type of synonym available to us is the public synonym. A **public synonym** is a unique synonym that when created becomes publicly available to all users. When a public synonym is created for an object, that synonym name is globally available and references the object for which it is created. We create public synonyms as shown in the next screenshot using the `CREATE PUBLIC SYNONYM` command:



Once created, the name `division_pub` can be used by any user in the database to reference the `companylink.division` table. Note that creating public synonyms requires special privileges in the database and is usually only done by database administrators. Also, remember that just because a synonym exists for an object, it does not mean that a user necessarily has permission to see the referenced table. A public synonym can be removed in the same way as a private one, except the command used is `DROP PUBLIC SYNONYM`.

SQL in the real world

Synonyms have certain benefits, but those benefits sometimes come into conflict with a company's coding standards. Many companies do not allow the use of synonyms, choosing instead to hardcode the schema name before every object. This allows anyone reading the code to know the specific schema in which the object resides. Other standards choose the ease of use that synonyms provide. Synonyms are also useful in situations where application code references objects that are frequently renamed. For instance, when a table name is changed, the administrator needs only to change the synonym's reference—not the code itself.

Summary

In this chapter, we've looked at a number of the different supporting objects in an Oracle database. We've seen how different types of indexes can be used to speed performance. We've looked at how sequences can be used to generate primary key values. We've examined the ways that views can simplify our SQL queries. Finally, we have seen how two different types of synonyms can aid our ability to easily name database objects.

Certification objectives covered

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

With the conclusion of this chapter, we've come to the end of the subject matter covered by the SQL Certified Expert test. But, we have one more chapter to go. Although optional for the test, the final chapter will help us complete our real-world ties to SQL. In it, we'll look at the ways that SQL is used inside programming languages such as Perl and Java. We'll also bring all the subjects we've learned together with several complex examples of SQL as a review.

Test your knowledge

1. Which of the following is the name given to a value that contains a pointer to the location of a row in the database?
    a. Sublanguage
    b. ROWID
    c. Bitmap index
    d. View
2. Which of the following is NOT a term associated with a B-tree index?
    a. Root nodes
    b. Leaf nodes
    c. Synthetic key nodes
    d. Branch nodes

3. Which of the following is a reason to create an index?
   a. You need to create a database view and a view requires an index
   b. You need to speed the performance of certain queries
   c. You need to generate primary key values
   d. You need to create an alternate name for a database table
4. Which of the following statements will successfully create a B-tree index?
   a. CREATE INDEX emp_idx ON employee (division_id);
   b. CREATE INDEX emp_idx ON employee;
   c. CREATE BITMAP INDEX emp_idx ON employee (division_id);
   d. CREATE INDEX emp_idx FOR employee (division_id);
5. Which of the following statements could be used to create a composite B-tree index?
   a. CREATE INDEX mess_idx ON message (message_text);
   b. CREATE INDEX mess_idx FOR message (message_text, message_date);
   c. CREATE INDEX mess_idx ON message (message_text, message_date);
   d. CREATE BITMAP INDEX mess_idx ON message (message_text, message_date);
6. Which of the following terms is used to describe a column with many distinct values, such as a primary key?
   a. High cardinality
   b. Low cardinality
7. Given a table column with 10,000 rows, which of the following values for selectivity would make it a good candidate for a bitmap index?
   a. 14
   b. 8
   c. 1
   d. 601
8. Which of the following statements could be used to create a bitmap index?
   a. CREATE BITMAP INDEX emp_bidx USING employee (gender);
   b. CREATE BITMAP INDEX emp_bidx ON employee (gender);
   c. CREATE INDEX emp_bidx ON employee (gender);
   d. CREATE INDEX emp_bidx ON employee (gender) TYPE (BITMAP);

9. Which of the following statements would correctly create a function-based index?
   a. CREATE INDEX mess_mon_idx ON message (to_char(message_date, 'MON'));
   b. CREATE FUNCTION BASED INDEX mess_mon_idx ON message (to_char(message_date, 'MON'));
   c. CREATE INDEX mess_mon_idx ON message (message_date, 'MON');
   d. CREATE INDEX mess_mon_idx ON message (message_date, 'MON') TYPE FUNCTION;

10. Which of the following statements could be used to rebuild an index?
    a. ALTER INDEX REBUILD mess_mon_idx;
    b. REBUILD INDEX mess_mon_idx;
    c. ALTER INDEX mess_mon_idx REBUILD IMMEDIATE;
    d. ALTER INDEX mess_mon_idx REBUILD;

11. Given a view has been created using the command below, what columns would be displayed when the following query is executed?

```
CREATE VIEW proj_vw
AS
SELECT project_name, project_mgr_id
FROM project;

SELECT * FROM proj_vw;
```

    a. project_name
    b. project_mgr_id
    c. project_name and project_mgr_id
    d. None. The creation of the view fails with an error

12. Which type of view can often be used to update values in a base table?
    a. Simple view
    b. Function-based view
    c. Complex view
    d. View created with the FORCE option

13. Given a sequence created with the following statement, what value would be generated for the branch_id column by the INSERT statement below?

```
CREATE SEQUENCE branch_seq
INCREMENT BY 10
START WITH 100;

INSERT INTO branch VALUES (branch_seq.NEXTVAL, 'Operations', 1);
```

    a. 14
    b. 100
    c. 110
    d. 120

14. What type of synonym would be created by the following statement?

```
CREATE SYNONYM division_syn FOR division;
```

    a. Private synonym
    b. Schema synonym
    c. Public synonym
    d. Sequence-based synonym

11
SQL in Application Development

We have reached the end of our journey through the SQL language. Although not required for the exam, by way of closure in this chapter, we look at some advanced topics in SQL. Later in the chapter, we will examine some SQL statements of a more challenging nature that combine many of the clauses we have seen. But, first, we examine the way that SQL is actually used in the real world. To do so, we'll look at some code snippets of various languages that embed SQL within them. It is important to understand that most of the code we will examine is not complete or functional in any way. This chapter is not meant to teach another programming language. Rather, it is a look "under the hood" at how SQL can be integrated with other languages. In closing, we'll take a look at some helpful hints and strategies for the taking the SQL certification exam.

In this chapter, we shall:

- Examine the ways SQL is incorporated into other programming languages
- Look at examples of SQL in third-generation languages
- Learn how the optimizer parses SQL statements
- Examine advanced SQL statements
- Review hints and strategies for taking the exam

Using SQL with other languages

In order to learn SQL, it has been necessary to look at it in isolation. This is the standard way to learn any programming language—place the user in "a vacuum" of sorts and use the language to solve various problems. This places the focus on the syntax and usage of the language while eliminating outside variables. Thus, we've performed our SQL operations using a tool such as SQL Developer against a single database. However, although this may be the most efficient way to *learn* SQL, it is not representative of the way that SQL is *used* in the real world. In an industry, when a requirement comes in to add a feature to an application that accesses the database, writing a statement in SQL Developer may constitute the beginning of the process, but it is certainly not the end of it. In real-world production, SQL code is not usually run from a client-side tool. More often than not, it is integrated within other programming languages as part of a more robust solution.

Why SQL is paired with other languages

To understand the way this works, we need to return full circle from where we began in *Chapter 1*, *SQL and Relational Database*. We mentioned that SQL is, like most programming languages, effective for some types of operations while ineffective for others. Computer applications are sometimes described in terms of *layers*. Generally, conventional wisdom states that there are three layers to any application—the data layer, the business logic layer and the presentation layer. While SQL is an effective language for manipulating the data layer, it would not be acceptable as a language for the presentation layer, due to its lack of functionality in that area. For this reason, in software development, SQL is usually paired with another programming language. SQL can be used to manipulate the data while other languages can be used to write the business logic and/or presentation layer. For instance, say that it has been decided that our *Companylink* application is going to be ported to a mobile phone platform. While we may use SQL for programming at the data layer, it is more likely that the presentation layer (and perhaps the logic layer) would be written in a language such as Java. SQL in itself is insufficient for this task, since it lacks many of the basic constructs crucial to programming languages, such as iteration (looping) and conditional statements (if...then). Fortunately, Oracle recognized this shortcoming and, in Oracle version 7, released the PL/SQL programming language.

Using SQL with PL/SQL

PL/SQL, or **Procedural Language SQL**, is Oracle's **third-generation language** (**3GL**) for extending SQL. Although it lacks some of the features present in modern languages, it is a true 3GL. While it is portable across platforms that use Oracle, it is proprietary to Oracle. PL/SQL allows for the storage of code in the form of database objects that are compiled and persistent within the database. PL/SQL adds many

of the operations typically found in programming languages, such as conditionals, iteration, and branching, to SQL. The two make a powerful, robust combination when working with Oracle database data. However, while PL/SQL can be used for the business logic layer along with SQL at the data layer, it has very few facilities that would make it appropriate for coding the presentation layer. PL/SQL is a block-structured language that is syntactically modeled on the Ada programming language. **Block structured** refers to the fact that PL/SQL constructs its major sections into blocks of code that work in concert to execute the program as desired. Let's examine a simple PL/SQL code snippet in the following code example:

```
CREATE PROCEDURE clink_ins_prc (
  p_div_id    number,
  p_div_name  varchar2
)
IS

BEGIN

INSERT INTO division
 VALUES (p_div_id, p_div_name);


COMMIT;


END;
```

Again, our purpose here is not an exhaustive exposition of PL/SQL; rather, to see the way that SQL is used within other languages. However, for clarity, let's point out some features of this code. First, the upper section (before `BEGIN`) is the definition of the object. This object, `clink_ins_prc`, is a stored procedure that is followed by two parameters. These parameters are values that are passed into the procedure during execution time. As we will see, it is parameters such as these that give our code flexibility and promote re-use. The second block (after `BEGIN`) is the body of the procedure. The body is where the action takes place. It is also here that we should recognize some familiar syntax. The body contains an `INSERT` statement followed by a `COMMIT`. However, notice that the values we send to the `INSERT` statement are not literals—they are the names of the parameters from the definition block. These values are passed to the procedure during runtime, which are then used in the `INSERT`. Once created, the procedure can be invoked as follows:

```
BEGIN
 clink_ins_prc (7, 'Public Relations');
END;
```

The procedure is invoked by its name, clink_ins_prc, and specifies two values to pass into the procedure: the number 7 and the string 'Public Relations'. These two values match the input parameters in the procedure itself. Thus, the number 7 is passed to the variable p_div_id, and the string 'Public Relations' is passed to the variable p_div_name. Once the body of the procedure is executed, these two values are active any time either of these variables is referenced. This offers us tremendous flexibility. Rather than simply retyping one INSERT statement after another, we could execute the procedure repeatedly, each time simply passing different values. PL/SQL can also utilize SELECT statements as well, as the following example demonstrates:

```
CREATE PROCEDURE clink_sel_prc (
  p_div_id  number
)

IS

lv_div_name varchar2(20);

BEGIN

SELECT division_name
 INTO lv_div_name
FROM division
WHERE division_id = p_div_id;


dbms_output.put_line('Division name is ' || lv_div_name);

END;
```

The structure of this procedure is very similar to the previous example. There is a declarative block that defines the procedure and any variables used, followed by the body of the code. Notice the SELECT statement used in the body. It is a simple SELECT of the type we've seen before. However, this statement also contains the keyword INTO. This is generally required of SELECT statements when using them with PL/SQL. The INTO keyword allows us to provide a variable, in this case lv_div_name, that contains our selected value. If we were to select multiple values, we would define and reference multiple variables—one for each value selected. As the keyword suggests, we are selecting values "into" variables. Once the value is contained in the variable, we can manipulate it as desired. In this case, we pass the variable to the dbms_output.put_line procedure that is used to print it on screen. We invoke our new procedure as follows:

```
BEGIN
 clink_sel_prc(7);
END;
```

In short, when we invoke `clink_sel_prc`, passing the value `7` to it, the procedure selects the `division_name` from the `division` table that has a `division_id` of `7` and prints the string "Division name is " followed by the value for `division_name`. Any time we invoke the procedure, we can pass any legal value for `division_id` and receive a different result.

SQL in the real world

It is very common to see PL/SQL in environments that use Oracle. Its tight integration with SQL gives it an extremely high-performing engine for the business logic layer of any data-intensive application. Often, applications are structured to use SQL at the data layer, PL/SQL at the logic layer, and a language such as Java for the presentation layer.

Using SQL with Perl

PL/SQL is somewhat unique in languages that use SQL. Since it was designed to be used with Oracle databases, its support for SQL is tightly integrated. It has many features that allow the code to directly access and manipulate SQL. Most programming languages are designed to be multi-purpose and have no such level of integration. For instance, consider the simple task of making a connection to the Oracle database. While PL/SQL requires no special coding to accomplish this, there is no similar "out of the box" functionality for many languages. Thus, even making a connection to the database would require a good deal of extra, low-level coding. What is needed is an *interface* between the language and the Oracle database. An interface would create a layer of abstraction between the two that would make it easier for the code to address the database. An interface, sometimes called an **API** or **Application Programming Interface**, provides a set of instructions or functions that facilitate interaction between a language and Oracle. Fortunately, due to the popularity of Oracle, many modern languages have just such interfaces.

Perl is a high-level, interpreted, general purpose language that has achieved a great deal of popularity owing to its relatively simple syntax, modularity, and compliance with the POSIX standard. Its wide use on the web for **CGI** (**Common Gateway Interface**) scripting has earned it the nickname, "the duct tape that holds the Internet together". It is also popular with DBAs and System Administrators for maintenance scripting. To connect to Oracle databases, programmers generally make use of the Perl **DBI** (**DataBase Interface**) and the **DBD::Oracle** module.

The DBI is a general interface used to connect to many different types of databases, and the DBD::Oracle module is used specifically for Oracle. Used together, they form the kind of interface between Perl and Oracle that programmers use for such operations as enabling web pages to talk to Oracle databases. A relatively simple example of this is shown in the following code:

```
#!/usr/bin/perl

my ($database, $user, $password, $div_name, $sql, $cursor);

use DBI;

$database='companylink';
$user='companylink';
$password='companylink';


$dbh=DBI->connect("dbi:Oracle:$database",$user,$password);

$sql="select division_name from division where division_id=7";


$cursor=$dbh->prepare($sql);
$cursor->execute();

while (($div_name)=$cursor->fetchrow_array) {
print "$div_name\n";
}
```

If you are unfamiliar with Perl, this may look confusing, so let's break it down a step at a time. The first section invokes the use of the DBI and defines variables that are prefixed with the `$` sign in Perl. The connection to the database is defined by the `DBI->connect` call, which receives the name of the database (`$database`), the username (`$user`), and the password (`$password`). Next, we assign a recognizable `SELECT` statement to the variable `$sql`. This is a convenient way to contain our statement for use later in the code. The actual execution of the statement is controlled by `$dbh->prepare` and `$cursor->execute`. Finally, the `while` statement fetches the result and places it in the `$div_name` variable for output to the screen.

Using SQL with Python

The **Python** programming language is, like Perl, a high-level, interpreted language. It continues to grow in popularity for its speed of execution, readable code, and object-oriented extensions. It is used for web development as well as standalone applications and integrates well with other languages, such as C++ and Java. Its large base of extensions makes it useful for coding the presentation layer as well.

Like Perl (and most languages), Python has no built-in support for manipulating Oracle databases. However, strong support in the open source community has provided a number of interfaces to Oracle. One of the most popular is the **cx_Oracle** extension module. It conforms to Python's API specification and allows Python code to interface with Oracle. A short piece of code using SQL and cx_Oracle is shown as follows:

```
import cx_Oracle

connection=
 cx_Oracle.connect("companylink","companylink","companylink")
cursor=connection.cursor()

cursor.execute
 ("select division_name from division where division_id=7")

v_div_name=cursor.fetchall()
```

Here, the connection string is established in the line beginning `connection=`. It references the cx_Oracle module and its connect method. We pass username, password, and database name, in that order, to the connect method. A `cursor` is declared using the connection, and our SQL statement is passed to its `execute` method. Our final line fetches the data into the `v_div_name` variable.

Using SQL with Java

In the past decade, few programming languages have taken the world by storm like **Java**. To call its use widespread would be an understatement. Developed originally in the early 1990s, it is used today in every kind of application, from client-side applications to dynamic web pages to cell phone apps. The motto of Java is "write once, run anywhere", owing to its platform independence. Code written in Java is run in a **JVM** or **Java Virtual Machine**, an abstraction layer that gives it widespread portability. While still considered a high-level language, Java is **object-oriented**, a programming paradigm that manipulates data structures known as *objects* in order to create programs. Oracle Corporation has historically taken a great interest in the Java programming language and has for many years taken steps to incorporate it into every level of its product offerings. In fact, Oracle assumed "stewardship" of Java with its acquisition of Sun Microsystems in 2010, although the core code of the language was made available under open source distribution terms in 2007. For Oracle professionals, Java has the distinction of being one of only three programming languages directly supported *within* the database, the other two being PL/SQL and SQL. Just as we can write PL/SQL-stored procedures as resident code within the database, we can do the same with Java-stored procedures.

Code written in Java generally makes use of the **JDBC** (**Java DataBase Connectivity**) interface to access Oracle (as well as other) databases. The JDBC interface, or *driver*, is invoked to make a database connection as shown in the following code snippet:

```
Connection connection = null;

try {

    String driver = "oracle.jdbc.driver.OracleDriver";
Class.forName(driver);

    String host = "clinkServer";
    String port = "1521";
    String sid = "companylink";

    String url =
     "jdbc:oracle:thin:@" + host + ":" + port + ":" + sid;

    String user = "username";
    String password = "password";

    connection =
DriverManager.getConnection(url, user, password);

} catch
...
```

Any connection to an Oracle database requires the name of the database server (host), the port on which Oracle runs (port), and the name of the database (**sid**, or **System IDentifier**). Our Java connection invokes the JDBC driver and passes the hostname, port, and sid to form a URL. That URL then receives the username and password to initiate the connection. Once the connection is made, our code need only invoke it and execute the query. The following example does this and fetches the result set into a variable:

```
try {

    Statement stmt = connection.createStatement();

   ResultSet rs =
     stmt.executeQuery("SELECT * FROM division");

}
```

Understanding the Oracle optimizer

When we execute a query in Oracle, it is easy to forget what's happening "under the hood". If we think about it for a moment, we might say that not all queries are created equal. For instance, when we discussed indexes, we noted that an index can speed up the execution of certain statements, provided that the `WHERE` clause is making its selection based on an indexed column. However, we noted that even when an index is present, an index scan may not be the most efficient execution path. What if we selected every row from a particular column in a table? In such a case, a full table scan might be more efficient. There are innumerable possibilities for such choices, given the scope and complexity possible with SQL. Clearly, there is something "under the hood" that is making these choices. In Oracle, it's called the optimizer. The **optimizer** is the engine that determines the most efficient manner in which to execute a SQL statement. It is a very complex internal component within Oracle that is constantly being improved with each release. Throughout its history, it has come in two different forms—the rule based optimizer and the cost based optimizer.

Rule-based versus cost-based optimization

The **rule based optimizer** is an older method for optimization that is no longer supported in Oracle 11g. Although it is not used in the current version, an understanding of how it works is useful to us in comprehending the concept of optimization. The rule based optimizer uses a set of 15 rules that are ranked in order of efficiency. The rules include query optimization methods such as full table scan, index scan, and scan by `ROWID`. These rules also take such factors as sorting, primary keys, and joins into account. Rule based optimization occurs when the optimizer *parses*, or interprets, the SQL statement and determines the rank for which it qualifies. Thus, if an index is present that matches the conditions of the query, it chooses to use the index.

While the rule based optimizer served adequately for many years, with the advent of larger databases and their high performance requirements it became clear that another method was needed. The complex queries written today require more flexibility in their choice of execution path. It is not enough that the optimizer chooses from a finite list of possible paths. Such factors as the size of the rows retrieved, the size of the table, and the structure of the index also need to be taken into account. Unlike the rule based optimizer, the **cost based optimizer** takes these and other factors into account. In short, the cost based optimizer determines the optimal execution path based not on a set of rules but on the *cost* of the query in terms of resource usage.

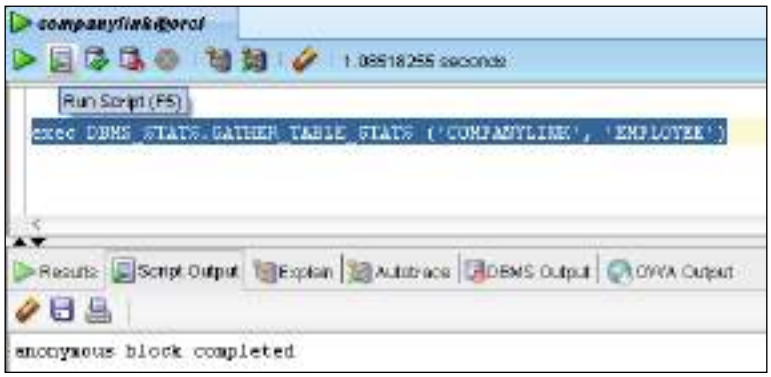
In order to make these types of choices intelligently, the cost based optimizer requires the collection of statistics. **Statistics** are gathered to quantify the characteristics of tables and their indexes. Statistics fall into four categories: table, column, index, and system. Table statistics include the number of rows in the table as well as their average length. Index statistics include the number of levels in an index. System statistics include disk I/O and CPU utilization. All these statistics provide vital information for the optimizer to choose the best optimization path.

Gathering optimizer statistics

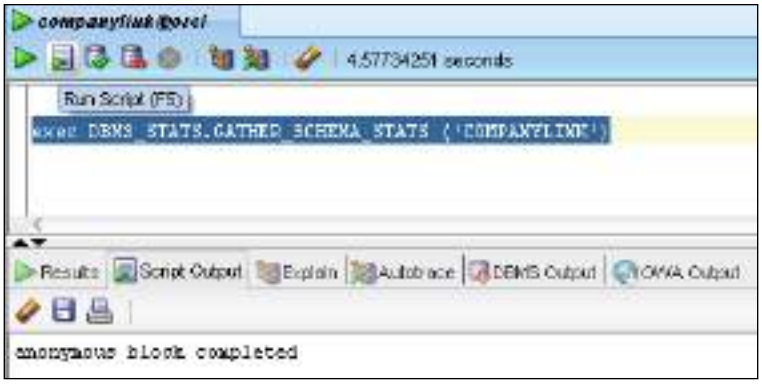
Just as the data and structure of tables and indexes change over time, so do their statistics. Take, for example, our `Companylink employee` table. As it is now, it contains 16 records and has (depending on how many examples you've completed) four indexes. What if we were to add 100,000 rows and two new indexes to the `employee` table? The characteristics of the larger table would be much different than before. For this reason, statistics must be *gathered*, or generated, periodically to ensure that the optimizer has the best possible information with which to make its decisions. When we gather optimizer statistics, we often refer to it as *analyzing* the table. The term comes from the older method of gathering statistics, which used a

statement called `ANALYZE`. In more recent versions of Oracle, statistics are gathered using one of two methods—manual or automatic.

We can analyze our tables manually using the `DBMS_STATS` package. `DBMS_STATS` is an Oracle-supplied PL/SQL package that allows us several options for statistics collection, including the ability to gather stats at the table or schema level and to take statistical samplings from tables. When we invoke the `DBMS_STATS` package in SQL Developer, we execute it using the **Run Script** button (or the *F5* key) next to the **Execute Statement** button that we normally use. Also, we must make sure we highlight the statement, since SQL Developer interprets PL/SQL slightly differently than SQL. This is shown in the following screenshot:



This operation scans the table and its indexes and records them as metadata within the database. The next time an SQL statement is run against the `employee` table, the cost based optimizer can make use of the statistics to determine the best possible execution path. We can also gather statistics on our entire `companylink` schema using the `GATHER_SCHEMA_STATS` procedure within `DBMS_STATS` as shown in the next screenshot. Remember to use the **Run Script** button.



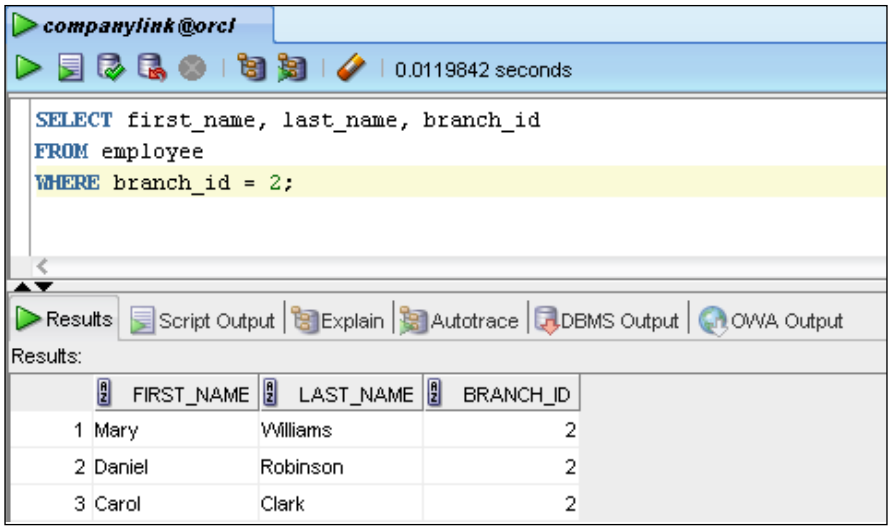
The lack of up-to-date statistics can be devastating to query performance. Because statistics are so vital, Oracle has developed a job that can automatically gather them for us. In version 10g, this was accomplished using the `GATHER_STATS_JOB`, an internal database job created by Oracle during database creation. In 11g, the automatic gathering of statistics is part of a maintenance task infrastructure known collectively as AutoTask. One of the surprising features of these automatic tasks is that they collect statistics intelligently. These tasks have the ability to determine which tables need statistics collection and which do not. For instance, say that Oracle automatically analyzes the `employee` table. The next time the job runs, AutoTask can determine whether the table has changed enough to warrant another collection. If the table is static over several days or weeks, Oracle can choose not to analyze, greatly reducing the amount of time needed to run the job.

SQL in the real world

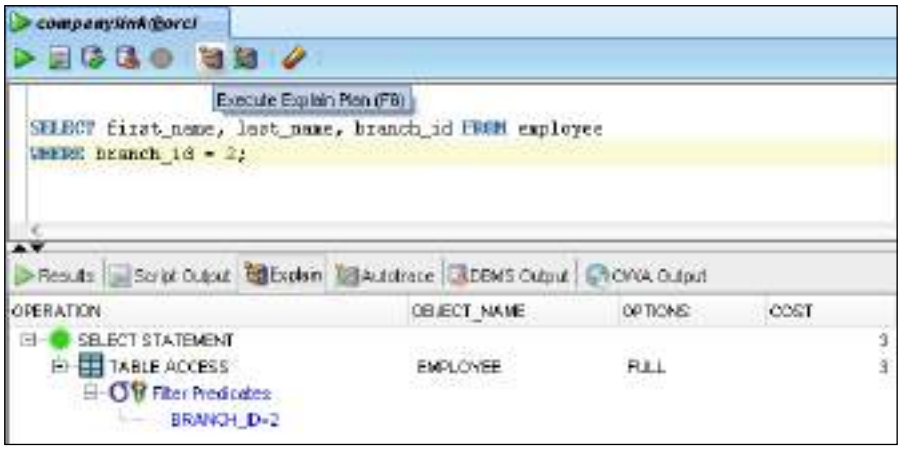
In large, highly active databases such as data warehouses, statistics collection can be extremely resource intensive. The statistics collection for a table with millions of rows can take hours to complete. In such situations, jobs to collect stats are done during times of low database usage and are sometimes done using a sample of records instead of the entire table.

Viewing an execution plan with EXPLAIN PLAN

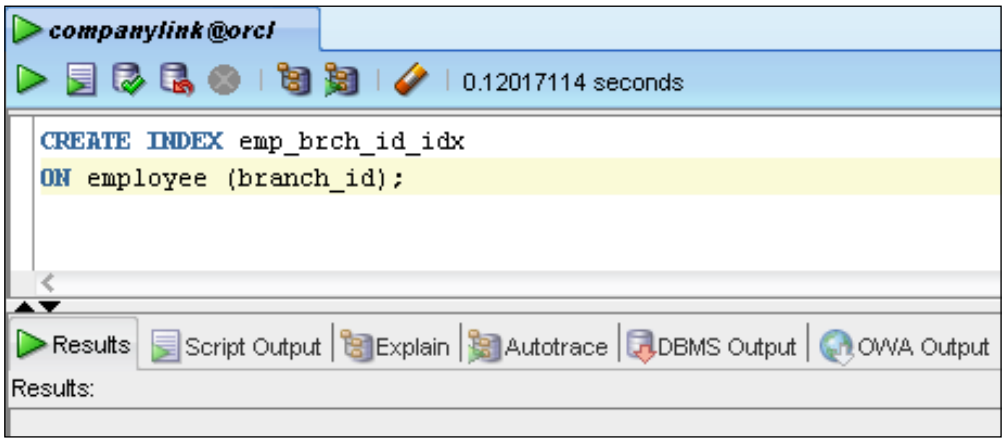
Now that we understand how the optimizer works, it is useful to see it in operation. The primary way we see into the "mind" of the optimizer is through the `EXPLAIN PLAN` statement. `EXPLAIN PLAN` will display an execution plan for a given statement, showing us what type of plan will be used, as well as the resource cost. While `EXPLAIN PLAN` is an actual SQL statement, today's GUI interfaces to SQL allow us to view our execution plan in a format that is much more readable than simply executing the statement itself. The tool we use to access the database will determine what steps we take to generate the plan, but the underlying method is the same. Since SQL Developer is our tool of choice, we will look at running explain plans in that tool. Let's first look at a simple SQL statement in the following screenshot and then look at its execution plan:



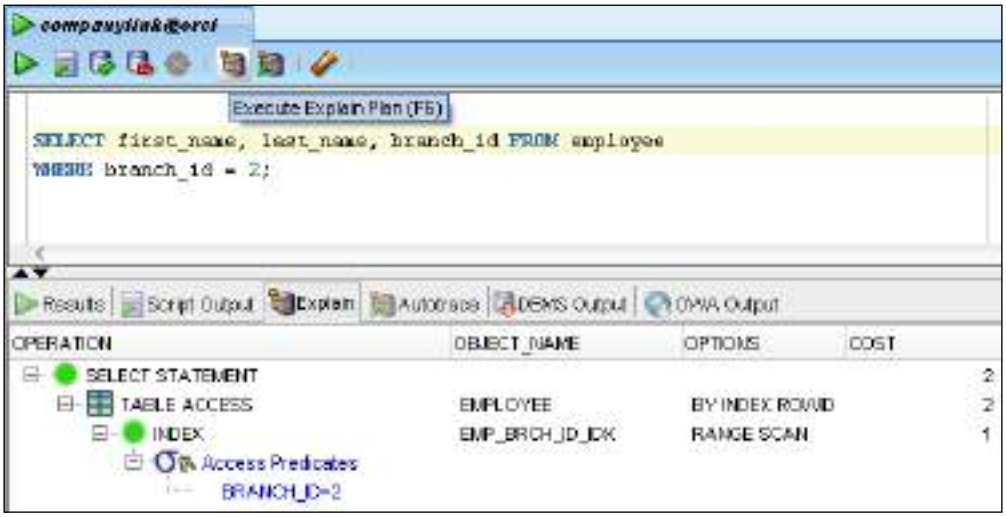
The query simply selects from the **employee** table, using a **branch_id** value of **2** as a limiting condition. From what we've learned of indexing, if an index is present on the `branch_id` column, it should be used. Next, we look at the `EXPLAIN PLAN` functionality as implemented in SQL Developer.



To execute our `EXPLAIN PLAN`, we click the **Execute Explain Plan** button (alternatively, the *F6* key) as indicated. The line **TABLE ACCESS** along with **EMPLOYEE** and **FULL** indicates that our execution plan will do a full table scan on the `employee` table. The **Filter Predicates** section shows our limiting condition—**BRANCH_ID = 2**. Also, a **COST** of **3** is shown for the query. In short, the execution plan indicates that a full table scan will be done to find rows with a `branch_id` value of `2`. As we know, full table scans are not the preferred access method when we are querying for a limited number of rows. But, we can only make use of an index scan if we have an index on the column with the limiting value, in this case `branch_id`. Let's add an index to that column and see the result.



Next, we view a different execution plan using the methods described previously.



As we can see, the type of table scan used now is **BY INDEX ROWID**, indicating an index scan is taking place with **EMP_BRCH_ID_IDX** shown as the name of the index being used. Our cost is also listed as **2**, indicating that the index scan requires less resources to execute. While the difference between the cost values may not seem significant, it is only because the `employee` table is very small. If the table in question was large, the cost difference would likely be substantial. Using `EXPLAIN PLAN`, we can see the benefits of using an index to speed performance.

SQL in the real world

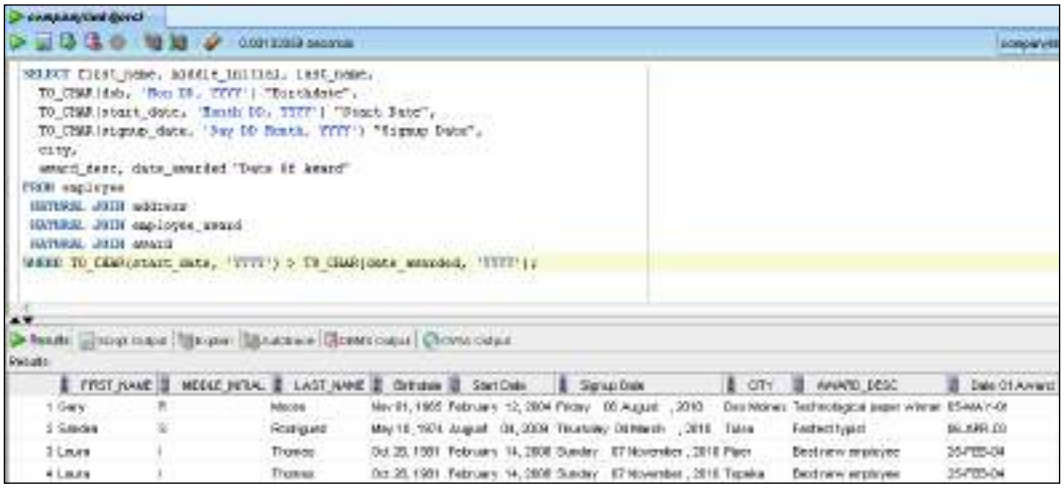
The importance of `EXPLAIN PLAN` cannot be overstated. Unfortunately, it is often overlooked by developers during the writing of SQL statements. Examining the execution plan of your SQL takes little effort and can reap enormous benefits in terms of performance. Taking the time to do so at the development stage can save hours of costly troubleshooting in production.

Advanced SQL statements

For our last look at SQL, we examine some statements of a more complex nature. It is important to note that these statements do not contain any elements that we haven't already covered. Their complexity derives from the number of elements they contain working together. Decoding and understanding these statements involves the ability to do the following three operations:

- Recognize the syntax being used
- Break the statements into component pieces
- Understand how the different pieces work together

Examining complex statements such as these is good practice for the SQL certification exam. The ability to master them will significantly increase your preparation for the test. Our first example is listed as follows:

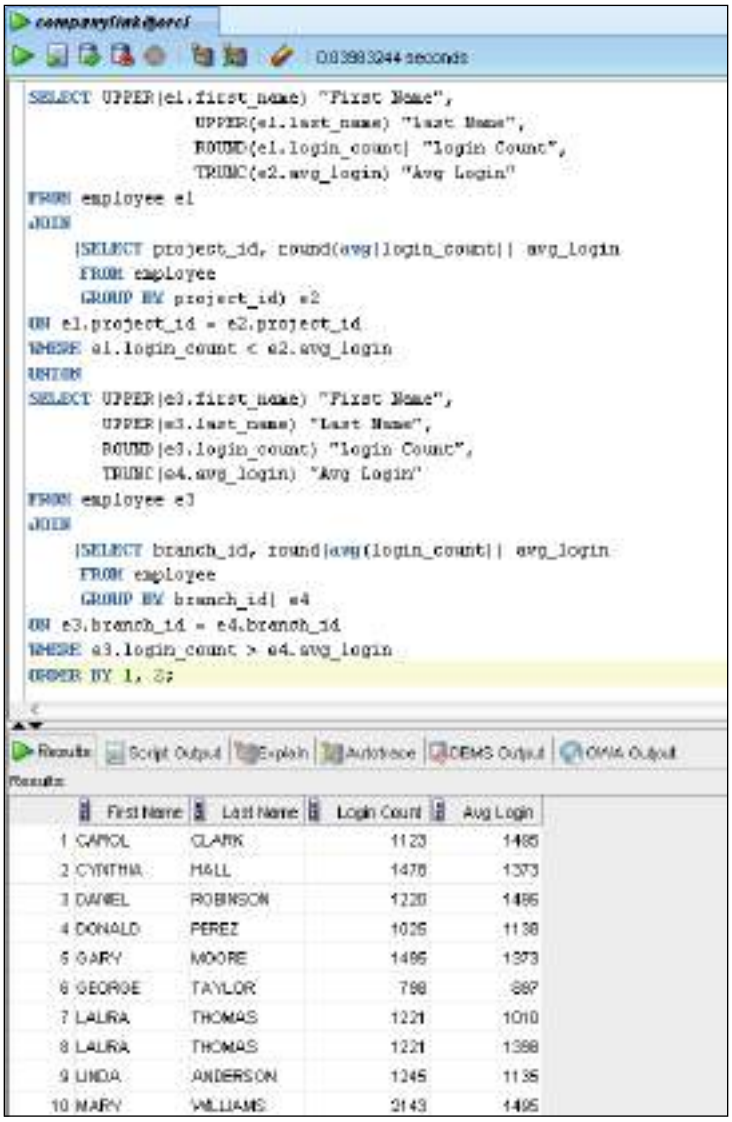


If we step back and take a broad look at this statement, we can see several pieces at work. The statement:

- Uses Oracle's join syntax (`NATURAL JOIN`)
- Joins four tables using three joins
- Does not have an overly complex `WHERE` clause, although it uses a function
- Uses a lot of formatting techniques, including date formatting and aliases

Provided that we don't allow ourselves to get lost in the syntax, we see that the statement is not as complex as it may appear. It joins columns from four tables and therefore uses three joins as we learned in our discussion of n-1 join conditions. These joins are natural joins, so the columns being joined are not explicitly specified. It may be necessary to look at the column structure of the tables involved to see what is occurring. The statement also uses the `TO_CHAR()` function to format the output of various dates, but it does so using different format masks. In short, the statement is a report that displays employee and address information for employees that have won awards. However, it is the brief `WHERE` clause that gives away the purpose of this statement. The `WHERE` clause restricts rows to those award winners whose start date as an employee is *after* the date of their award. Since an employee cannot receive an award if they haven't yet been hired, the report is designed to reveal inaccuracies in how award dates are stored.

After viewing this report, we can correct the dates using `UPDATE` statements. Take a look at our next example.



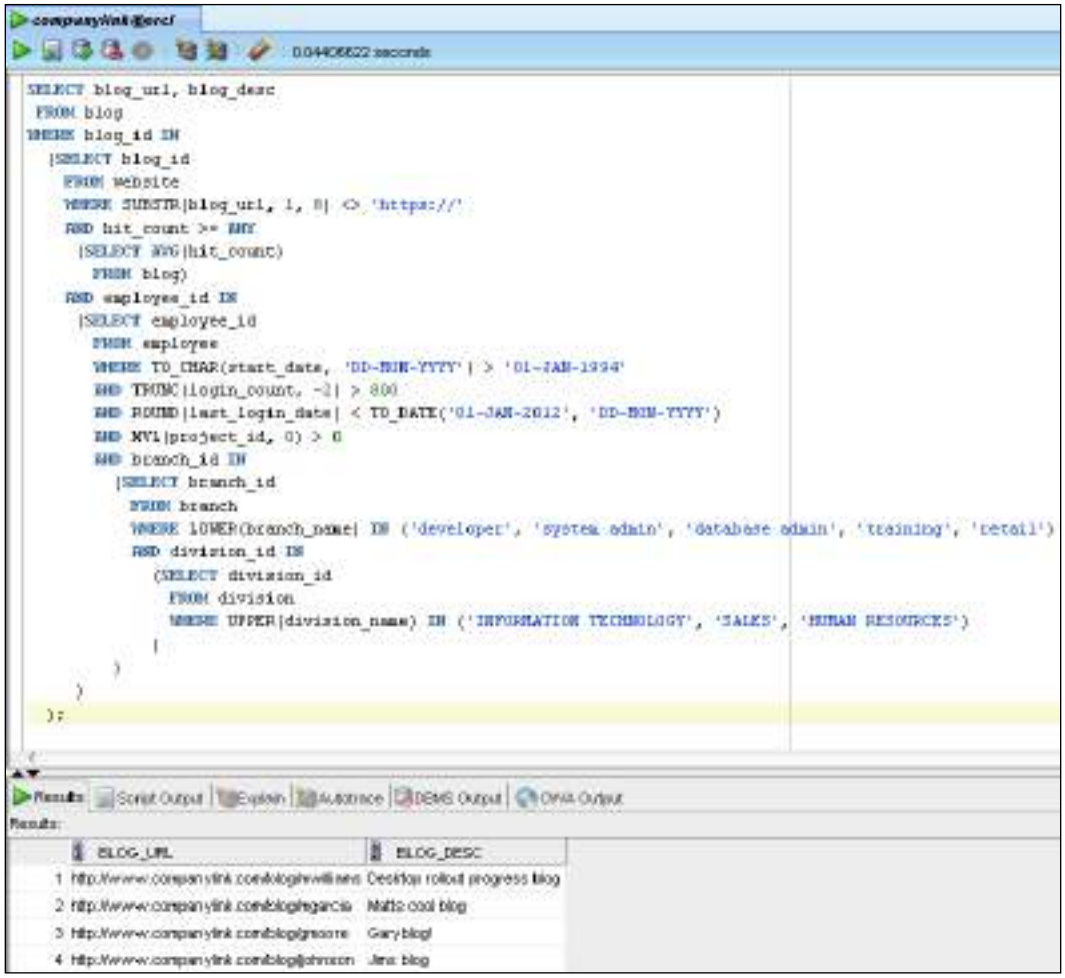
```
SELECT UPPER(e1.first_name) "First Name",
               UPPER(e1.last_name) "Last Name",
               ROUND(e1.login_count) "login Count",
               TRUNC(e2.avg_login) "Avg Login"
FROM employee e1
JOIN
    (SELECT project_id, round(avg(login_count)) avg_login
     FROM employee
     GROUP BY project_id) e2
ON e1.project_id = e2.project_id
WHERE e1.login_count < e2.avg_login
UNION
SELECT UPPER(e3.first_name) "First Name",
       UPPER(e3.last_name) "Last Name",
       ROUND(e3.login_count) "login Count",
       TRUNC(e4.avg_login) "Avg Login"
FROM employee e3
JOIN
    (SELECT branch_id, round(avg(login_count)) avg_login
     FROM employee
     GROUP BY branch_id) e4
ON e3.branch_id = e4.branch_id
WHERE e3.login_count > e4.avg_login
ORDER BY 1, 2;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	First Name	Last Name	Login Count	Avg Login
1	CAROL	CLARK	1123	1485
2	CYNTHIA	HALL	1478	1373
3	DANIEL	ROBINSON	1220	1486
4	DONALD	PEREZ	1025	1138
5	GARY	MOORE	1486	1373
6	GEORGE	TAYLOR	798	887
7	LAURA	THOMAS	1221	1010
8	LAURA	THOMAS	1221	1388
9	LINDA	ANDERSON	1245	1135
10	MARY	WILLIAMS	2143	1495

While the preceding statement consists of many lines of SQL, we can begin to dissect it immediately by splitting it in half. Notice the **UNION** operator roughly in the middle of the code. This tells us that we are actually dealing with two distinct SQL statements and then uniting their output with **UNION**. Once we've recognized this, we also notice the similarities between the two statements. Both use inline views joined with the **employee** table. In fact, the two statements are almost identical except for the grouping in the inline views and the slightly different conditions in the **WHERE** clauses. In short, this statement looks at two slightly different dimensions of the data; one grouped by project and one by branch, and then displays the two together.



Our last statement is certainly a long one. Again, while it may look too complex to tackle, it's not. The previous statement contains five subqueries on four sub-levels, since two of the subqueries occur at the same level. The first thing we must do when we encounter a complex, nested subquery is to start at the innermost subquery. In this case, the innermost query is a relatively simple one that selects values meeting a certain condition from the `division` table. It then passes those values back to the next level—the branch level—a query that uses the values from the first query as conditional input. The resulting values are passed back to the employee level, where the values are processed along with several other query conditions involving dates. These results return to the website level; a level with two subqueries. The first queries the `blog` table for average hit count, and the second is the resulting rollup of the other nested subqueries we've discussed. All of these values return to the originating query that displays blog information for the values returned by all of the nested subqueries. In short, this statement displays a fairly inefficient substitute for a join of the `blog`, `website`, `employee`, `branch`, and `division` tables done with subqueries. Joins can often be rewritten as subqueries, but because of the way the Oracle optimizer processes joins, it is often very inefficient to do so.

Exam preparation

In this book, we've laid out the subjects needed to take the Oracle Database 11g: SQL Fundamentals I exam. In order to pass, you'll need to achieve a high level of competence in these areas. However, there are some helpful hints that we pass on here to help you on your way.

Helpful exam hints

- **Understand the basics:** The exam, number 1Z0-051, is given as a proctored test at a Pearson VUE testing center. Once you're ready to take the test, you can register online and choose a testing center near you for convenience. The cost for the exam is $125. You are allowed 120 minutes to complete 70 multiple choice questions. In order to pass, you must have a score of 60 percent, or 42 questions correct. Also, visit Oracle's certification website at `http://certification.oracle.com`, and click the **Certification** link for more information on this and other tests in the Oracle certification track.
- **Don't cut corners**: Passing the test is all about preparation. Where possible, don't drag out your preparation over a long period of time. Try to set aside a period of time that you will spend in dedicated preparation, and then take the test. Do NOT skip subjects. Your exam will be 70 questions generated from a larger bank of questions. While it is likely that not *every* subject will be on your exam, *any* subject is possible. Don't risk skipping any particular subject.

- **Know your syntax**: This exam is heavily oriented toward SQL syntax. It is **crucial** that you know proper syntax to pass the exam. There are many questions designed to exploit common syntax errors. We have included many syntactically-oriented questions at the end of each chapter to help prepare you for this.
- **Learn by doing:** Most people learn a subject more thoroughly by actually doing it. Take advantage of the sample database and examples used in this book. This can take you a long way toward having a quicker recognition of many questions.
- **Supplement your learning**: This book covers all the necessary subjects to pass the exam, but don't hesitate to supplement your knowledge from other sources. Take advantage of practice exams and sample questions where possible. Search the Internet—some sites have forums visited by people who have taken the test. Their comments may give you some insight into whether you're ready to take the exam. However, beware of "braindump" types of practice exams. Many have inaccurate questions and answers.
- **Don't sweat it:** Be mentally and physically prepared on the day of the exam. Eat properly and get plenty of rest. Prepare well and be confident. Many candidates like to schedule the exam in the earlier part of the day so they're mentally fresh. Choose a testing site that you're familiar with to minimize the risk of being late. Stress is a candidate's worst enemy.
- **Watch the wording**: Certification tests in general can be notorious for the wording of their questions, and Oracle's are no exception. Read each question carefully. Testing candidates generally do not have trouble completing the SQL exam within the time allotted provided they are properly prepared. Watch out for poorly worded questions and double negatives. The exam is multiple choice and some questions have many correct answers. While you must get every one of the answers correct for full credit on a question, partial credit is awarded for each correct choice.
- **Use the questions to your advantage**: There is no penalty for guessing on the exam, so make sure you answer all the questions. Also, you have the opportunity to skip questions and return to them later. The answer on one question may help jog your memory on another. When you read a question, try to deduce the solution without immediately referring to the possible answers. If you think you're right, don't let the answers sway you. Some candidates like to skim the test and complete the "easier" questions first. This can give you some momentum for the remainder of the test.

A recommended strategy for preparation

While passing the certification test is certainly the goal, earning a certification without understanding the subject matter is useless. Many "braindumps" and "exam crams" attempt to push a candidate through the certification process, mostly on memorization. Even if a candidate does pass, they are left without the requisite knowledge to function in their certified field. This book attempts to actually teach you proficiency in the subjects you need and includes many examples to prepare you for using SQL in the real world. To that end, we recommend that you take a real-world approach in preparing for the test. It may take longer than simply cramming for the exam, but when you're finished, you'll be much better prepared.

After you've finished this book, take a "dry run" at the test. Without reviewing, go back and answer the questions in this book and any example questions you may have from other sources. When you check your answers, make note of the questions you've missed, as well as their subject areas. From this, you have a list of subjects on which you need to focus. This prevents you from spending an inordinate amount of time on the subjects you already know. Using your list of review subjects, go back through the book and re-read the sections that cover these areas. Work through the examples again. Write your own variations to these examples. For instance, say that after completing your dry run, you find that you have a good grasp of the basics of using subqueries, but you struggle with multi-column subqueries. Read through that section in *Chapter 8*, *Combining Queries* and do the examples. Then, see if you can modify the examples to write your own multi-column subqueries. Once you are done with your review of the subject areas, take the test questions again. Repeat the process of studying and working with the remaining subject areas. Use this iterative process to fully prepare you for every subject. Once you feel you're ready, take the test. Good luck!

Summary

In our final look at SQL, we've examined some of the issues that are important in real-world SQL development. We've seen how SQL can be integrated into other programming languages, such as Perl and Java. We examined the Oracle optimizer and how statistics gathering and `EXPLAIN PLAN` can be used to produce efficient SQL code. We've looked at several complex SQL statements that utilize many of the aspects of SQL we have learned and broken them down into logical, more manageable components. Finally, we have laid out a preparation strategy for taking the SQL certification exam and looked at some helpful hints.

A

Companylink Table Reference

Considering the amount of work that we do in this book with our `Companylink` tables, it can be advantageous to have a place of reference to see the tables used in our fictitious Companylink application. This appendix details the tables, columns, and datatypes used as the basis for the `Companylink` application.

The Companylink data model

The next screenshot shows the tables that are included in the `Companylink` data model. The tables are created and populated by running the `companylink_db.cmd` Windows command file. This file calls the `companylink_ddl.sql` script that creates the table structures and the `companylink_data.sql` script that populates them with data. These are the base tables used for examples throughout this book. Other tables are added through the course of various chapters, and modifications to them are made as a part of the examples.

ADDRESS

The `address` table contains the addresses for employees that are a part of the `Companylink` application.

Column Name	Data Type	Nullable	Default	Primary Key
ADDRESS_ID	NUMBER(10,0)	No	-	1
STREET_ADDRESS	VARCHAR2(50)	Yes	-	-
CITY	VARCHAR2(25)	Yes	-	-
STATE	VARCHAR2(2)	Yes	-	-
ZIP	NUMBER(5,0)	Yes	-	-
EMPLOYEE_ID	NUMBER(10,0)	Yes	-	-

1 - 6

AWARD

The `award` table contains the description for various awards given to `Companylink` employees.

Column Name	Data Type	Nullable	Default	Primary Key
AWARD_ID	NUMBER(10,0)	No	-	1
AWARD_DESC	VARCHAR2(4000)	Yes	-	-
				1 - 2

BLOG

The blog table contains data on the blogs hosted by the `Companylink` application.

Column Name	Data Type	Nullable	Default	Primary Key
BLOG_ID	NUMBER(10,0)	No	-	1
BLOG_URL	VARCHAR2(250)	Yes	-	-
BLOG_DESC	VARCHAR2(4000)	Yes	-	-
HIT_COUNT	NUMBER(10,0)	Yes	-	-
				1 - 4

BRANCH

The `branch` table contains information regarding the company branch to which employees are assigned.

Column Name	Data Type	Nullable	Default	Primary Key
BRANCH_ID	NUMBER(10,0)	No	-	1
BRANCH_NAME	VARCHAR2(50)	Yes	-	-
DIVISION_ID	NUMBER(10,0)	Yes	-	-
				1 - 3

DIVISION

The `division` table contains information regarding the company division to which employees are assigned.

Column Name	Data Type	Nullable	Default	Primary Key
DIVISION_ID	NUMBER(10,0)	No	-	1
DIVISION_NAME	VARCHAR2(100)	Yes	-	-
				1 - 2

EMAIL

The `email` table contains e-mail address information for `Companylink` employees.

Column Name	Data Type	Nullable	Default	Primary Key
EMAIL_ID	NUMBER(10,0)	No	-	1
EMAIL_ADDRESS	VARCHAR2(50)	Yes	-	-
EMPLOYEE_ID	NUMBER(10,0)	Yes	-	-
				1 - 3

EMPLOYEE

The `employee` table contains information on the various employees who are a part of the `Companylink` application.

Column Name	Data Type	Nullable	Default	Primary Key
EMPLOYEE_ID	NUMBER(10,0)	No	-	1
FIRST_NAME	VARCHAR2(25)	Yes	-	-
MIDDLE_INITIAL	VARCHAR2(1)	Yes	-	-
LAST_NAME	VARCHAR2(50)	Yes	-	-
GENDER	CHAR(1)	Yes	-	-
DOB	DATE	Yes	-	-
START_DATE	DATE	Yes	-	-
BRANCH_ID	NUMBER(10,0)	Yes	-	-
PROJECT_ID	NUMBER(10,0)	Yes	-	-
SIGNUP_DATE	DATE	Yes	-	-
LAST_LOGIN_DATE	DATE	Yes	-	-
LOGIN_COUNT	NUMBER(10,0)	Yes	-	-
				1 - 12

EMPLOYEE_AWARD

The `employee_award` table is a bridge table between the `employee` and `award` tables that indicates how awards have been assigned to employees.

Column Name	Data Type	Nullable	Default	Primary Key
AWARD_ID	NUMBER(10,0)	Yes	-	-
DATE_AWARDED	DATE	Yes	-	-
EMPLOYEE_ID	NUMBER(10,0)	Yes	-	-
				1 - 3

MESSAGE

The `message` table contains information about the messages that have been sent through the `Companylink` application.

Column Name	Data Type	Nullable	Default	Primary Key
MESSAGE_ID	NUMBER(10,0)	No	-	1
MESSAGE_TEXT	VARCHAR2(4000)	Yes	-	-
MESSAGE_DATE	DATE	Yes	-	-
EMPLOYEE_ID	NUMBER(10,0)	Yes	-	-

1 - 4

PROJECT

The `project` table contains information about the various projects to which `Companylink` employees have been assigned.

Column Name	Data Type	Nullable	Default	Primary Key
PROJECT_ID	NUMBER(10,0)	No	-	1
PROJECT_NAME	VARCHAR2(250)	Yes	-	-
PROJECT_DESC	VARCHAR2(4000)	Yes	-	-
PROJECT_MGR_ID	NUMBER(10,0)	Yes	-	-

1 - 4

WEBSITE

The `website` table contains data on the websites hosted by the `Companylink` application.

Column Name	Data Type	Nullable	Default	Primary Key
WEBSITE_ID	NUMBER(10,0)	No	-	1
WEBSITE_URL	VARCHAR2(250)	Yes	-	-
WEBSITE_DESC	VARCHAR2(4000)	Yes	-	-
BLOG_ID	NUMBER(10,0)	Yes	-	-
HIT_COUNT	NUMBER(10,0)	Yes	-	-
EMPLOYEE_ID	NUMBER(10,0)	Yes	-	-

1 - 6

B

Getting Started with APEX

Throughout this book, we have used the SQL Developer tool for our SQL statement examples. This appendix covers the use of another tool that can be used—Oracle Application Express, or APEX.

Oracle Application Express

While the SQL Developer tool used in this book is a robust tool for writing and executing SQL, it has the disadvantage of requiring the reader to have direct access to a database. If you already have a database that you can use to run the examples, you need not concern yourself with using APEX. However, if you do not, you would need to install the Oracle software and create a database before being able to connect using SQL Developer. While we briefly covered the steps in making a database connection in *Chapter 1, SQL and Relational Databases*, installing Oracle and creating databases is outside the scope of this book. Doing so also requires you to have a server that can be used to host your database. The Oracle database software can be freely downloaded for personal use to do this and can run on many modern home PCs. However, in the event that you do not have a machine available to do this, APEX can be used to do the examples in this book without the resources needed to run an entire database.

What is APEX?

Oracle Application Express (APEX) is a free, hosted service provided by Oracle that provides a workspace for users that can be used to create database objects and run SQL statements without installing Oracle on your local machine. In essence, it is a free private database that you can create and access from a web browser. Strictly speaking, APEX is a standalone product offered by Oracle, but the company hosts a website that runs APEX and allows users to create accounts to evaluate the product.

Signing up for APEX

To use APEX, we must first sign up for a free account. We navigate our browser to: `http://apex.oracle.com` and are presented with the following APEX login screen:

The first time we use APEX, we need to sign up for an account by clicking **Sign Up**. We're presented with the welcome screen that follows, and then we click **Next**.

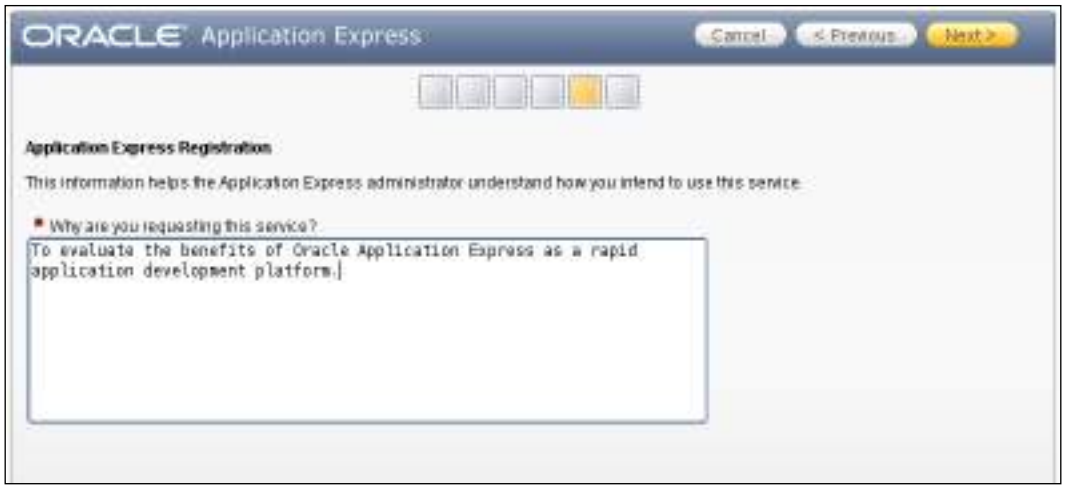
The next screen asks us for our first and last name, along with our e-mail address. A verification message will be sent to this address, so we need to make sure our e-mail is valid.

When you sign in to APEX, you will be asked to provide the name of this workspace, which is filled in the **Workspace** field as shown in the following screenshot. The **Username** (e-mail address), **Password**, and **Workspace** name must all be entered when you login to APEX each time, so don't forget them.

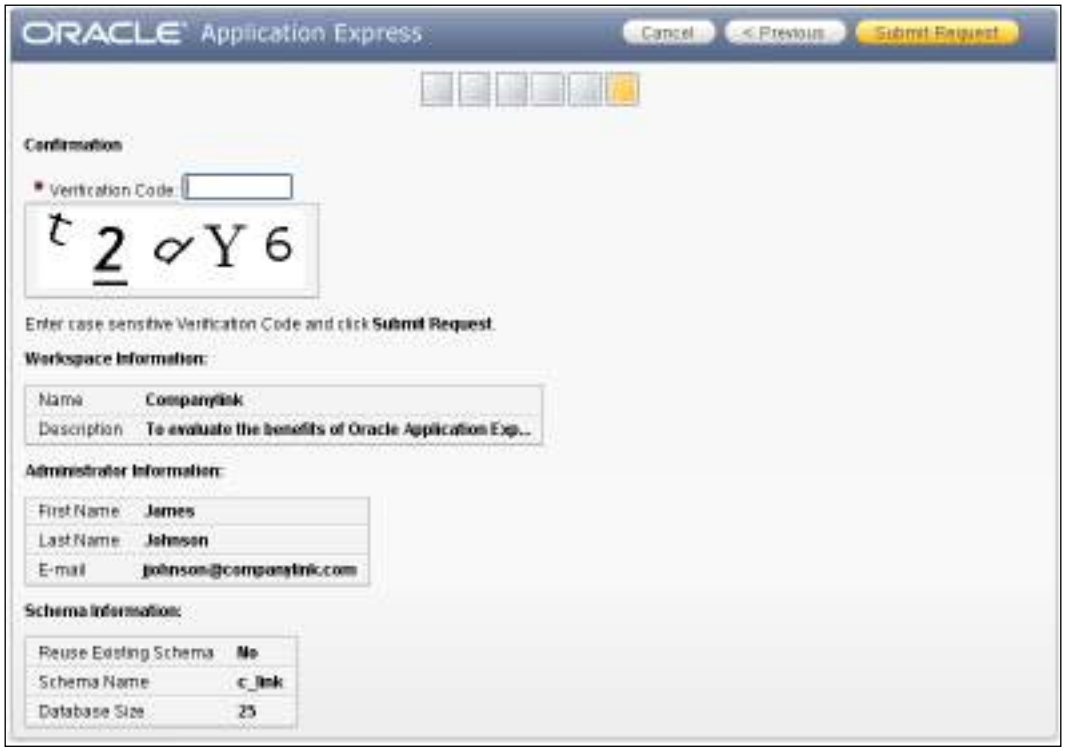
The next screen asks whether we should use an existing database schema or create a new one. Our first workspace will require a new schema, so we want to make sure the **Request a new schema** button is selected.

Since we have chosen to create a new schema, we provide a name for it on the following screen. Choose a name that is appropriate for you. We can also choose an initial space allocation for storage, but the default of 25 MB should be fine for our examples.

The next screen asks us to provide a reason we're requesting access to the APEX service. Since this is a free service, this seems only fair. This request may actually be reviewed by a real person, so we need to make an appropriate request, as shown in the following screenshot:



Finally, we type in the verification code and click **Submit Request**. An e-mail will be sent to the address you used in your request. Follow the instructions in that e-mail to finalize your account.



Using APEX

Getting around in APEX is very intuitive. APEX is a virtual playground for an aspiring SQL programmer. It allows you to do the following:

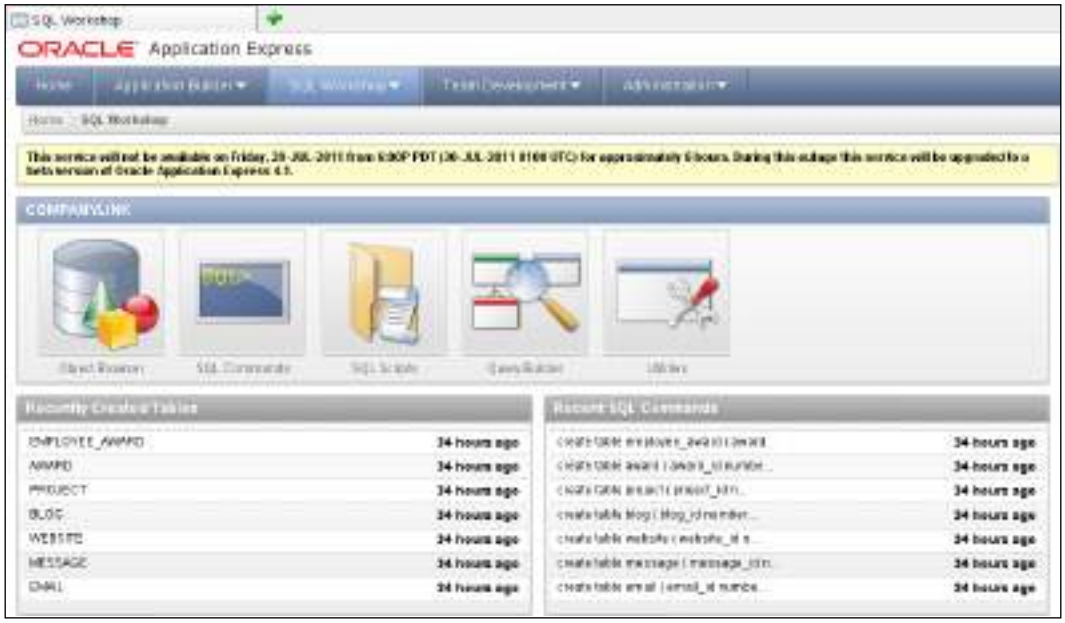
- Create your own tables, indexes, constraints, sequences, and so on
- Run SQL statements and scripts
- Build PL/SQL objects

There are many other capabilities of APEX for Oracle professionals. We can use the GUI **Application Builder** to create our own web applications. We can do team development of projects. We can even load and unload data quickly and easily. As an interesting note, nearly all the prototyping of the SQL statements used in this book was done using APEX. While it lacks some of the features of SQL Developer, it is easy to use.

To begin using APEX, we return to the APEX home page at `http://apex.oracle.com`. Click the **Login** button, and enter your account information.

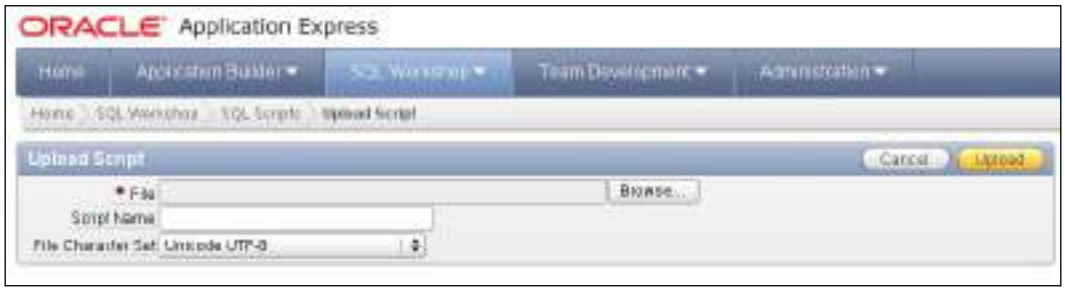
The home page for APEX is shown in the next screenshot. Since our focus is on SQL development, we click the box for **SQL Workshop**.

We can choose from a number of options from the SQL Workshop page. The **Object Browser** will allow us to view our database objects through a user friendly GUI. We will use the **SQL Commands** page to execute our queries. But, first, since we opened a new account, we need to create the database objects used in this book. To do this, we click **SQL Scripts.**

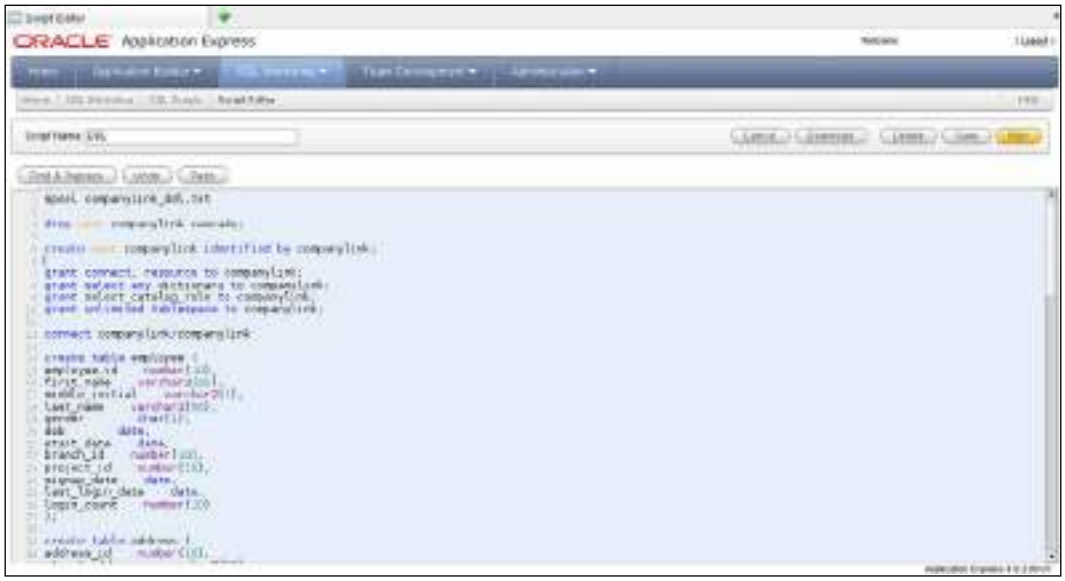


Next, we click **Upload** to upload new scripts. Browse to the following two scripts that create the `Companylink` data and click **Upload** for each one. These scripts can be downloaded from the Packt website:

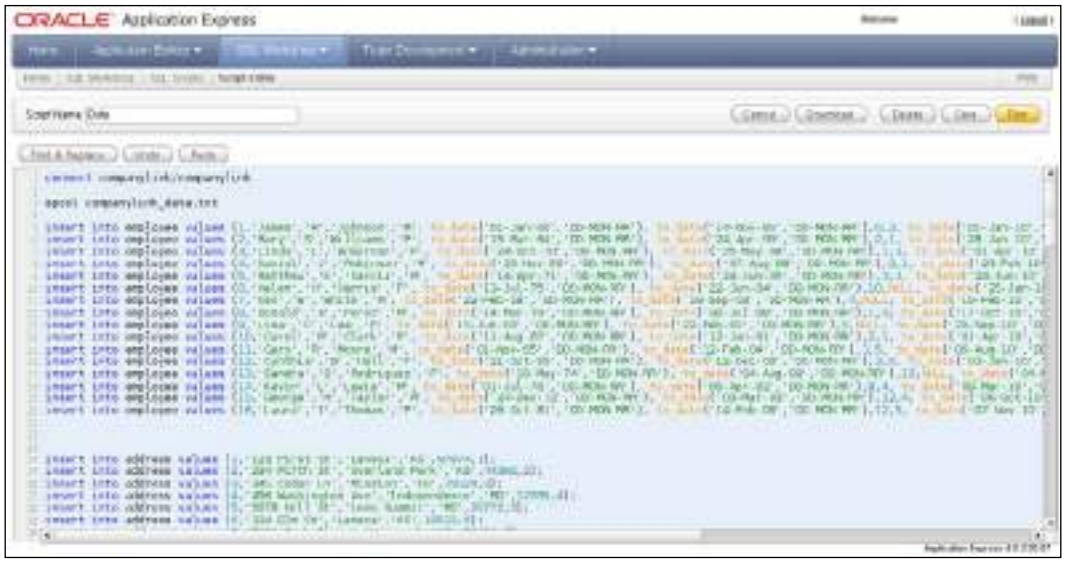
- `companylink_ddl.sql`
- `companylink_data.sql`



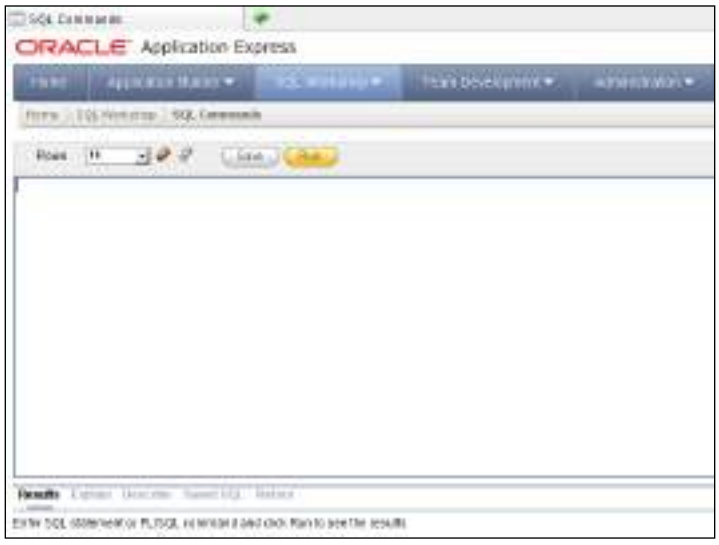
Once we've returned to the SQL Scripts page, we click to pencil icon (under **Edit**) next to our `companylink_ddl.sql` script. This script creates the table structures used in this book. The code is then previewed for us.



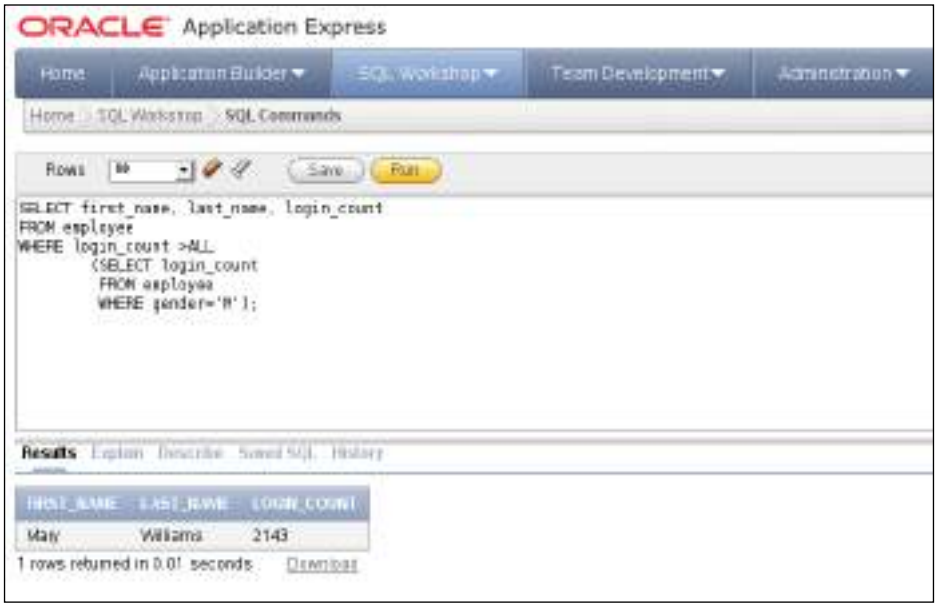
We simply click **Run** to execute the script. We may see a message that some commands are ignored, since they are needed for SQL Developer but not for APEX. Once this is complete, we do the same for the `companylink_data.sql` script that populates our tables with example data.



If we wish, we can also do this for the `companylink_constraints.sql` script. Once the scripts have run successfully, click **SQL Workshop** on the top blue bar to return to the workshop page. Click the **SQL Commands** button.



Here, we simply enter any SQL statements we wish, and click the **Run** button to execute it. In the following example, we use one of the SQL subquery statements we completed in *Chapter 8*, *Combining Queries*.



Any other query can be and run in the same way. When using APEX, remember that there are a few differences from SQL Developer. First, the default format for dates is DD/MM/YYYY. This requires us to make some alterations to the example code we enter when using the `TO_CHAR` and `TO_DATE` functions. Second, some functions, such as LPAD and RPAD, don't format row output in the same way as SQL Developer. Lastly, some formatting options are unavailable in APEX. However, beyond these few differences, we should be able to use APEX for the majority of the examples in this book. Also, don't be afraid to explore some of the other features of APEX. They can be both educational and fun.

Index

Symbols

(+) symbol 169
1NF 11
3GLs 18
3NF 14
4GL 18
4NF 14
5NF 14

A

ACID test
and transaction control 139
ADD_MONTHS() function 233
ADDRESS, Companylink data model 419
aggregate functions 244
alias notation
two table joins, using 165, 166
ALL
using, with multi-row subqueries 282-286
alternative naming
synonyms, used 388
ALTER TABLE
used, for modifying tables 329
ALTER TABLE... DROP COLUMN
used, for removing columns 335, 336
ALTER TABLE... MODIFY
used, for changing column characteristics 332-334
American National Standards. *See* **ANSI**
American Standard Code for Information Interchange (ASCII) 84
ANSI 18
ANSI standard
versus Oracle proprietary syntax 158
ANSI standard joins
about 159
cartesian joins 160, 161
data retrieving from multiple tables, n-1 join conditions used 171-175
equi joins 162
full outer join 170
inner joins 166, 167
left outer join 168
non-equi join 176, 177
outer join 168
right outer join 170
self-join 177
structure 159
syntax 159
ANY
using, with multi-row subqueries 282-286
APEX
about 423
home page, URL 428
Request a new schema button 426
Run button 432
signing up for 424-426
SQL Commands button 431
SQL Commands page 429
Upload button 430
using 428-432
API 401
Application Programming Interface. *See* **API**
arithmetic functions
MOD() function 230
ROUND() function 227-230
TRUNC() function 229, 230
using 227

arithmetic operators
using, with SELECT statement 53
ASC
used, for changing sort order 104, 105
AVG() function 251, 252, 279
AWARD, Companylink data model 420
award_id column 51
award table 119

B

balanced tree structure 364
BETWEEN clause
about 87
used, for constructing range conditions 86-88
bitmap indexes
about 369
cardinality 369, 370
creating 371
structure 370, 371
block structured 399
BLOG, Companylink data model 420
Boolean AND operator 97
Boolean conditions
in WHERE clause 94
Boolean NOT operator 98-100
Boolean operators 94
Boolean OR operator 95, 96
BRANCH, Companylink data model 420
Branch_ID 107
B-tree indexes
creating 366-368
B-tree structure 364

C

cardinality 369, 370
cartesian join
about 160, 161
using, with cross join 178, 179
cartesian product 160, 161
case conversion functions
using 199
case conversion functions, string functions
INITCAP() function 202
LOWER() function 202
UPPER() function 200, 201
CGI 401
character datatype errors
avoiding 318-322
CHAR datatype 310
CHECK constraint 348, 349
adding 352
code beautifier 24
column alias 165
column, RDBMS 17
columns
adding, to tables 329-331
characteristics changing, ALTER TABLE... MODIFY used 332-334
joining, JOIN USING used 184-186
joining, NATURAL JOIN used 180-182
removing, ALTER TABLE... DROP COLUMN used 335, 336
comma-separated values (CSV) 9
COMMIT
transactions, completing with 140-142
Common Gateway Interface. *See* **CGI**
Companylink database
about 25, 38, 95
creating 26
Companylink data model
about 419
address table 419
award table 420
blog table 420
branch table 420
CHECK constraint, adding 352
Companylink model, tables adding to 353-355
Companylink tables, constraints adding to 349
division table 420
email table 421
employee_award table 421
employee table 421
message table 422
NOT NULL constraint, adding 352
project table 422
referential integrity, adding 350-352
website table 422

Companylink tables
constraints, adding 349
complex view
about 379
and simple view, distinguishing 378-380
composite B-tree indexes 368
composite primary key 344
CONCAT() function 208, 209
conditional inserts 125-127
conditions, WHERE clause
about 80-85
equality conditions 80, 81
non-equality conditions 82-85
conditions, with multiple values
about 86
ampersand substitution, using with runtime conditions 101, 102
Boolean AND operator 97
Boolean conditions, in WHERE clause 94
Boolean NOT operator 98-100
Boolean OR operator, examining 95, 96
pattern-matching conditions, LIKE clause used 91-94
range conditions constructing, BETWEEN clause used 86-88
set conditions creating, IN clause used 89, 90
correlated subqueries
using, with multi-row subqueries 287, 289
correlated subquery 287
cost based optimizer 406
COUNT() function 245, 246
CREATE INDEX command 366
CREATE TABLE statement
creating 314
cross join
cartesian join, using with 178, 179
CRUD model
and DML 117
CTAS
used, for copying tables 326, 328
CURRENT_TIMESTAMP
and SYSDATE, distinguishing 217-219
cx_Oracle extension 403

D

data
accessing, from multiple tables 156-158
creating, with INSERT 118
retrieving from multiple tables, n-1 join conditions used 171-175
retrieving, SELECT statements used 42
data analysis 243
database constraint
about 338
CHECK constraint 348, 349
FOREIGN KEY constraint 345-347
NOT NULL constraint 339-341
PRIMARY KEY constraint 341-344
UNIQUE constraint 348
database object 308
database user 308
data integrity
about 338
CHECK constraint 348, 349
enforcing, database constraints used 339
FOREIGN KEY constraint 345-347
NOT NULL constraint 339-341
PRIMARY KEY constraint 341-344
UNIQUE constraint 348
values, deleting with referential integrity 347, 348
Data Manipulation Language (DML) 117
data transformation
single-row functions, using 198
datatype
about 309
CHAR datatype 310
DATE datatype 313
NUMBER datatype 312, 313
VARCHAR2 datatype 311
datatype errors
avoiding 318
character datatype errors, avoiding 318-321
numeric datatype errors, avoiding 322-326
date arithmetic functions
about 231
ADD_MONTHS() function 233
MONTHS_BETWEEN() function 232
DATE datatype 313

date functions
about 217
characters converting to dates, with TO_DATE() function 223, 224
datatype conversion functions, utilizing 219
date to character conversion, using with TO_CHAR 219, 220
numbers converting, TO_NUMBER() function used 224-227
SYSDATE and CURRENT_TIMESTAMP, distinguishing 217-219
DBArtisan 22
DBArtisan XE. *See* **DBArtisan**
DBD$$Oracle module 401
DBI (DataBase Interface) 401
DDL 308
DECODE() function 236, 237
DELETE statement
data removing unconditionally, with TRUNCATE 136-138
purpose 133
rows, deleting by condition 133-135
rows, deleting without limiting condition 135
syntax 133
DESC
used, for changing sort order 104, 105
DESCRIBE command
about 317
used, for describing table structure 48, 49
DISTINCT
used, for displaying unique values 62-64
DISTINCT clause 76
DIVISION, Companylink data model 420
DML
and CRUD model 117
dob column 81
domain key. *See* **natural key**
DROP TABLE
used, for removing tables 337
DUAL table 54-56

E

EMAIL, Companylink data model 421
email_copy 124
EMPLOYEE_AWARD, Companylink data model 421
EMPLOYEE, Companylink data model 421
Entity Relationship Diagram. *See* **ERD**
equality conditions 80, 81
equi joins
about 162
two table joins, implementing with table-dot notation 162-164
two table joins, using with alias notation 165, 166
ERD 171
execution plan
viewing, with EXPLAIN PLAN 408-411
EXPLAIN PLAN
used, for viewing execution plan 408-411

F

fifth normal form. *See* **5NF**
first normal form. *See* **1NF**
flat file databases 8, 9
flat file paradigm
limitations 9, 10
FOREIGN KEY constraint 345-347
fourth-generation language. *See* **4GL**
fourth normal form. *See* **4NF**
FROM clauses
about 77
multi-column subqueries, using 291, 292
full outer join 170
function-based indexes 372, 373
functions
about 197
ADD_MONTHS() function 233
case conversion functions 199
CONCAT() function 208
DECODE() function 236
INITCAP() function 202, 203
INSTR() function 212
LENGTH() function 204
LOWER() function 202
LPAD() function 206
LTRIM() function 208
MOD() function 230
MONTHS_BETWEEN() function 232

multiple-row functions 199
NVL2() function 235
NVL() function 234
principles 198
ROUND() function 227, 229
RPAD() function 206
RTRIM() function 208
single-row functions 199
single-row functions, for data transformation 198, 199
string functions 199
SUBSTR() function 209
TO_CHAR function 219
TO_DATE() function 223
TO_NUMBER() function 224
TRUNC() function 229
UPPER() function 200

G

gender column 82
GROUP BY clause
about 276
data, grouping with 254, 255
pitfalls, avoiding 256-259
GROUP BY function
extending 260-262
grouping data
about 252, 253
GROUP BY function, extending 260-262
GROUP BY, pitfalls avoiding 256-259
principles 244
row group exclusion, HAVING clause used 263-265
statistical functions, using 262
with GROUP BY 254, 255

H

HAVING clauses
multi-row subqueries, using 286, 287
row group exclusion, performing 263-265
scalar subqueries, using 277
heap-organized tables 364
hit_count 83, 276

I

IN clause
about 90, 91
used, for creating set conditions 89, 90
using, with multi-row subqueries 280-282
indexes
about 364
bitmap indexes 369
B-tree indexes 364, 365
B-tree indexes, creating 366- 368
composite B-tree indexes, using 368
dropping 374
function-based indexes 372, 373
modifying 374
Oracle ROWID 362, 363
table scan 362
using, to increase performance 361
INITCAP() function 202
inline constraints 344
inner joins 166
INSERT
data, creating with 118
INSERT..SELECT statement 124
INSERT statement
about 400
conditional insert 125
multi-row inserts 124
named column notation, using 121, 122
NULL values, using 122, 123
positional notation, using 119, 120
single table inserts, using 119
syntax 118
INSTR() function 212-214
International Organization for Standardization. *See* **ISO**
INTERSECT set operator 298, 299
ISO 18, 158

J

Java
SQL, using 404
Java DataBase Connectivity. *See* **JDBC**
Java Virtual Machine. *See* **JVM**
JDBC 404

joining tables, principles
about 155
ANSI standard versus Oracle proprietary syntax 158
data, accessing from multiple tables 156-158
JOIN ON
used, for constructing fully-specified joins 186-189
joins
constructing, JOIN ON used 186-189
JOIN USING
used, for building multi-table joins 190, 191
used, for joining explicit columns 184-186
JVM 404

K

keywords 42

L

last_login_date value 85, 87
last_name column 84
left outer join 168
LENGTH() function 204, 205
LIKE clause
about 91, 94
used, for pattern-matching conditions 91-94
login_count value 282
LOWER() function 202
LPAD() function 206, 207
LTRIM() function 208

M

many-to-many relationship 14
mathematical operators
with SELECT 57-59
MAX() function 248-250, 277, 284
MESSAGE, Companylink data model 422
message_text column 309
MIN() function 248-250
MINUS set operator 299
MOD() function 230
MONTHS_BETWEEN() function 232
multi-column subqueries
using, with FROM clauses 291, 292
using, with WHERE clauses 290
multi-column UPDATE statements
writing 131, 132
multiple columns
selecting, from table 44-46
multiple tables
data, accessing from 272
multi-row functions
about 199, 244
AVG() function 251, 252
COUNT() 245, 246
in SQL 244
MAX() function 248-250
MIN() function 248-250
SUM) function 250, 251
multi-row inserts 124
multi-row subqueries
about 280
ALL, using 282-286
ANY, using 282-286
correlated subqueries, using 287, 289
IN, using 280-282
multiple rows, processing 280
using, with HAVING clauses 286, 287
multi-table joins
building, with JOIN USING 190, 191
multi-table natural joins
creating 190

N

n-1 join conditions
used, for retrieving data from multiple tables 171-175
n-1 join conditions, writing
multi-table joins building, JOIN USING used 190, 191
multi-table natural joins, creating 190
Oracle syntax used 189
named column notation 121
naming
alternative naming, synonyms used 388
object naming, synonyms used 386
private synonym, creating 388-390
public synonyms, creating 391
schema naming 387

NATURAL JOIN
used, for joining columns 180-182
natural key
versus synthetic key 345
nested functions 214, 215
nested functions, string functions
about 214
values, substituting with REPLACE() function 216
nesting subqueries 292, 293
non-correlated subquery 287
non-equality conditions 82-85
non-equi join 176, 177
normalization 10
NOT NULL constraint 339-341
adding 352
NULL 60, 61
NULL values
subqueries, using 294, 295
NUMBER datatype 312, 313
numeric datatype errors
avoiding 322-325
NVL2() function 235
NVL() function 234

O

object naming
synonyms used 386
Object Relational Database Management System. *See* **ORDBMS**
optimizer statistics
gathering 406, 407
Oracle Application Express. *See* **APEX**
Oracle join syntax
about 178
benefits 184
cartesian joins, using with cross join 178, 179
columns joining, NATURAL JOIN used 180-182
explicit columns joining, JOIN USING used 184-186
fully-specified joins constructing, JOIN ON used 186-189
used, for writing n-1 join conditions 189
Oracle optimizer
about 405
cost based optimizer 406
rule based optimizer 406
Oracle proprietary syntax
versus ANSI standard 158
Oracle SQL Developer
about 24
benefits 24
connection name 30
hostname 30
password 30
port 30
save password 30
setting up 27-30
username 30
ORDBMS 15
order
order changing, DESC used 104, 105
ORDER BY clause 75, 103
outer join 168
out of line constraint 344
output
formatting, SELECT statement used 50-53

P

pattern-matching conditions
LIKE clause used 91-94
Perl
SQL, using with 401, 402
persistent storage
and CRUD model 116
principles 116
PL/SQL
SQL, using with 398
PL/SQL Developer 24
positional method 118
positional sort 109
PRIMARY KEY constraint 341, 342, 344
primary keys
generating, sequences used 382-386
private synonym 388-390
Procedural Language SQL. *See* **PL/SQL**
PROJECT, Companylink data model 422
pseudo-table 55
public synonyms 391

Python
SQL, using 403

Q

query 42

R

range conditions
constructing, BETWEEN clause used 86-88
raptor. *See* **Oracle SQL Developer**
RDBMS
about 8
column 17
flat file databases 8, 9
flat file paradigm, limitations 9, 10
normalization 10-12
relational approach 13, 14
tables 16
relational approach, RDBMS 13
Relational Database Management Systems. *See* **RDBMS**
relational databases
languages for 18
relational paradigm 10
relational theory
and set theory, co paring 297
REPLACE() function 216
Request a new schema button 426
right outer join 170
ROLLBACK command
about 144
used, for undoing transactions 142-146
ROUND() function 227-230, 279
ROWID 362
RPAD() function 206, 207
RTRIM() function 208
rule based optimizer 406
Run button 432
Run Script button 145

S

scalar subqueries
about 274, 275
using, with HAVING clauses 277
using, with SELECT clauses 278, 279
using, with WHERE clauses 275-277
schema naming 387
SELECT
mathematical operators 57-59
SELECT clauses
about 77, 103
scalar subqueries, using 278, 279
selective views
creating 377, 378
SELECT statement
about 318
all columns, selecting from table 46-48
arithmetic operators, using 53, 55
columns, projecting 42
data, retrieving with 42
multiple columns, selecting from table 44-46
output formatting, aliases used 50-53
single column, selecting from table 43, 44
table structure displaying, DESCRIBE used 48, 49
SELECT statements
values, concatenating in 65-69
self-join 177, 178
sequences
about 345, 382
using, to generate primary keys 382-386
set conditions
creating, IN clause used 89, 90
set operators, SQL
INTERSECT set operator 298, 299
MINUS set operator 299
UNION ALL set operator 300, 301
UNION set operator 300
set theory
and relational theory, comparing 297
intersect 296
intersection of A and B 296
principles 296
sid (System IDentifier) 30, 405
signup_date 85
simple view
about 379
and complex view, distinguishing 378-380
single-column UPDATE statements
writing 128-131

single-row functions
about 199
for, data transformation 198, 199
single-row subqueries 274, 275
single table inserts
named column notation, using 121, 122
NULL values, using 122, 123
positional notation, using 119, 120
using 119
Solid State Disks (SSD) 116
sort
order changing, ASC used 104, 105
secondary sorts 106
sorting data
ORDER BY clause, used 103, 104
secondary sorts 106-108
sort order changing, ASC used 104, 105
sort order changing, DESC used 104, 105
SQL
about 7, 18
American National Standards Institute (ANSI) 18
case-sensitive 39, 40
Companylink database 25, 26
DBArtisan (DBArtisan XE) 22
DBArtisan XE (DBArtisan) 22
fourth-generation languages (4GL) 18
in real world 18, 19
language, for regional databases 18, 19
multi-row functions, using 244
Oracle SQL Developer 24
pairing, with other languages 398
PL/SQL Developer 24
purpose 38
SQL*Plus 20
SQL Worksheet 23
statement terminators 41, 42
syntax 38, 39
third-generation languages (3GLs) 18
Tool for Oracle Application Developers (TOAD) 21, 22
tools 20
using 398
using, with Java 404, 405
using, with Perl 401, 402
using, with PL/SQL 398-400
using, with Python 403
whitespace, uses 40
SQL Commands button 431
SQL Commands page 429
SQL Developer. *See* **Oracle SQL Developer**
SQL*Plus 20
SQL statements
advanced 411- 415
SQL Worksheet 23
standard deviation 262
statement terminators 41
statistical functions
STDDEV() function 262
using 262
VARIANCE() function 263
statistics 406
STDDEV() function 262
string functions 199
string literals
uses 54-56
string manipulation functions, string functions
CONCAT() function 208, 209
INSTR() function 212, 213, 214
LENGTH() function 204
LPAD() function 206
RPAD() function 206
RTRIM() and LTRIM() function 208
SQL, writing with 203
SUBSTR() function 209, 210, 211
Structured Query Language. *See* **SQL**
subqueries
data, accessing from multiple tables 272
issues, resolving 272-274
multi-column subqueries 289
multi-row subqueries 280
nesting subqueries 292
principles 271
scalar subqueries 274, 275
single-row subqueries 274, 275
types 274
using, with NULL values 294, 295
SUBSTR() function 209, 211
SUM() function 250, 251
surrogate key. *See* **synthetic key**
synthetic key
about 345
versus natural key 345

SYSDATE
and CURRENT_TIMESTAMP, distinguishing 217

T

table alias 165
table-dot notation
two table joins, implementing 162-164
table, RDBMS 16
tables
columns, adding 329-331
copying, CTAS used 326, 328
creating 315-317
modifying, ALTER TABLE used 329
removing, DROP TABLE used 337
table scan 362
table structure
displaying, DESCRIBE used 48, 49
TCL sublanguage, commands
COMMIT 139
ROLLBACK 139
SAVEPOINT 139
third-generation languages. *See* **3GLs**
third normal form. *See* **3NF**
TOAD 21
TO_CHAR function 219- 222
TO_DATE() function
used, for converting character dates 223, 224
TO_NUMBER() function
used, for converting numbers 224-227
Tool for Oracle Application Developers. *See* **TOAD**
transaction control
about 138
and ACID test 139
transactions
completing, with COMMIT 140-142
undoing, ROLLBACK used 142-144
TRUNCATE command
about 337
data, removing unconditionally 136-138
TRUNC() function 229, 230

U

Undo tablespace 143
union 296
UNION ALL set operator 300, 301
UNION set operator 300, 301
UNIQUE constraint 348
updatable view 378
UPDATE statement
multi-column UPDATE statements, writing 131, 132
purpose 128
single-column UPDATE statements, writing 128-131
syntax 128
UPPER() function 200, 201

V

VARCHAR2 datatype 311
VARIANCE() function 263
view
changing 381
creating 375- 377
options, configuring 381
selective views, creating 377, 378
simple and complex views, distinguishing 378-380
viewing 381

W

WEBSITE, Companylink data model 422
WHERE clause
Boolean conditions 94
conditions 80-85
multi-column subqueries, using 290
scalar subqueries, using 275-277
syntax 76-78
whitespace, uses 40

Z

zip code 94

Thank you for buying

OCA Oracle Database 11g: SQL Fundamentals I: A Real-World Certification Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Enterprise

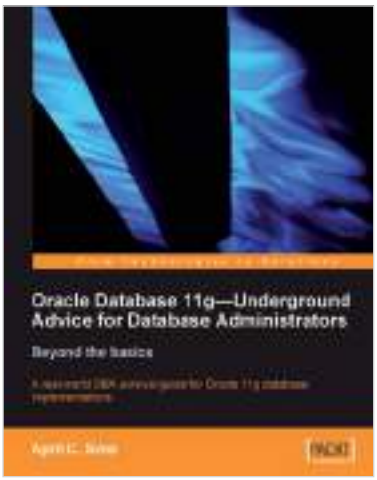
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING



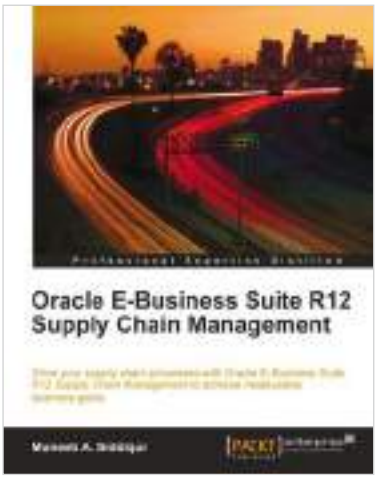
Oracle Database 11g – Underground Advice for Database Administrators

ISBN: 978-1-849680-00-4 Paperback: 588 pages

A real-world DBA survival guide for Oracle 11g database implementations

1. A comprehensive handbook aimed at reducing the day-to-day struggle of Oracle 11g Database newcomers
2. Real-world reflections from an experienced DBA—what novice DBAs should really know
3. Implement Oracle's Maximum Availability Architecture with expert guidance

---



Oracle E-Business Suite R12 Supply Chain Management

ISBN: 978-1-84968-064-6 Paperback: 292 pages

Drive your supply chain processes with Oracle E-Business R12 Supply Chain Management to achieve measurable business gains

1. Put supply chain management principles to practice with Oracle EBS SCM
2. Develop insight into the process and business flow of supply chain management
3. Set up all of the Oracle EBS SCM modules to automate your supply chain processes

[PACKT] enterprise
professional expertise distilled
PUBLISHING

Oracle Fusion Middleware Patterns

ISBN: 978-1-847198-32-7 Paperback: 224 pages

10 unique architecture patterns enabled by Oracle Fusion Middleware

1. First-hand technical solutions utilizing the complete and integrated Oracle Fusion Middleware Suite in hardcopy and ebook formats
2. From-the-trenches experience of leading IT Professionals
3. Learn about application integration and how to combine the integrated tools of the Oracle Fusion Middleware Suite - and do away with thousands of lines of code

---

Getting Started with Oracle Hyperion Planning 11

ISBN: 978-1-84968-138-4 Paperback: 620 pages

Design, configure, and implement a robust planning, budgeting, and forecasting solution for your organization using Oracle Hyperion Planning

1. Successfully implement Hyperion Planning—one of the leading planning and budgeting solutions—to manage and coordinate all your business needs with this book and eBook
2. Step-by-step instructions taking you from the very basics of installing Hyperion Planning to implementing it in an enterprise environment
3. Test and optimize Hyperion Planning to perfection with essential tips and tricks