Disclaimer

This course provides an overview of features and enhancements planned in release 11g. It is intended solely to help you assess the business benefits of upgrading to 11g and to plan your IT projects.

This course in any form, including its course labs and printed matter, contains proprietary information that is the exclusive property of Oracle. This course and the information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This course and its contents are not part of your license agreement nor can they be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This course is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remain at the sole discretion of Oracle.

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS
The U.S. Government’s rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.
Index
A

Alias 1-17
ALL Operator 7-19, 8-16, 8-17
ALTER TABLE Statement 10-35
Alternative Quote (q) Operator 1-23
American National Standards Institute (ANSI) i-30, 4-4, 6-5
Ampersand Substitution 2-29, 2-30, 2-33
AND Operator 2-16, 2-21, C-12
ANY Operator 7-18
Arithmetic Expressions 1-11, 1-15, 1-19
Arithmetic Operators 1-11, 1-12, 3-26
Attributes i-23

B

BETWEEN Operator 2-10
BI Publisher i-14

C

Cartesian Product 6-33, C-5
CASE Expression 4-37, 4-38
Character strings 2-7
CHECK Constraint 10-27
COALESCE Function 4-32, 4-33, 4-34
Column Alias 1-17
Comparison Operators 2-8, 2-9
Concatenation Operator 1-20
Constraints 9-4, 10-2, 10-16, 10-17, 10-18, 10-19, 10-29, 10-30, E-6
Conversion Functions i-5, 3-7, 3-11, 3-12, 3-28, 4-1, 4-4,
4-9
COUNT Function 5-9
CREATE SEQUENCE Statement 11-25
CREATE TABLE Statement 10-7
Creating a Database Connection i-37, i-38, i-39, i-58
Cross Joins 6-34
CURRENT_DATE 3-24, 9-9
CURRVAL 10-9, 10-27, 11-3, 11-22, 11-27, 11-28, 11-29, 11-33, 11-40
D
Data Types 4-28, 10-12, 10-13, 10-14
Database i-2, i-3, i-4, i-8, i-9, i-10, i-11, i-12, i-13,
   i-14, i-15, i-16, i-17, i-18, i-19, i-27, i-28, i-29, i-30, i-33,
   i-35, i-37, i-38, i-39, i-40, i-47, i-49, i-50, i-53, i-54, i-55,
   i-56, i-58, i-59, 1-14, 1-15, 3-4, 3-5, 3-10, 3-16, 3-24, 4-9,
   4-27, 5-27, 6-2, 6-6, 7-8, 9-3, 9-13, 9-15, 9-19, 9-21, 9-25,
   9-26, 9-27, 9-31, 9-39, 9-40, 9-42, 10-3, 10-4, 10-5, 10-6, 10-11,
   10-14, 10-15, 10-17, 10-31, 10-34, 10-36, 10-37, 10-38, 11-4, 11-6, 11-16,
   11-26, 11-28, 11-31, 11-35, 11-37, 11-42, 11-43, C-2, D-17, D-19, E-3,
   E-13, E-15, E-19, E-20
Database Transactions 9-26, 9-27
Date i-18, 1-9, 1-21, 1-33, 2-24, 2-31, 3-3, 3-5, 3-7, 3-8,
   3-15, 3-20, 3-22, 3-23, 3-24, 3-27, 3-28, 3-29, 3-31, 3-33, 4-12,
   4-13, 4-14, 4-22, 9-10, 10-12, 10-14
Datet ime Data Types 10-14
DBMS i-17, D-17, D-19
DECODE Function 4-39, 4-40, 4-41
DEFAULT Option 10-9
DELETE Statement 9-21
DESCRIBE Command 1-27
DISTINCT Keyword 5-10
DUAL Table 3-17
Duplicate Rows 1-24

E
Entity Relationship i-21, i-22, i-23, B-3
Equijoins 6-12, 6-35, C-9, C-10, C-11, C-22
Execute SQL D-5, D-20
Execute Statement icon i-44, i-47, 1-8, 1-30, 9-48, 10-40
Explicit Data Type Conversion 4-7, 4-8, 4-9

F
FOR UPDATE clause 9-3, 9-13, 9-19, 9-25, 9-39, 9-42, 9-43, 9-44,
   9-46
Format Model 4-12, 4-14
| F | Functions i-5, 2-7, 3-1, 3-2, 3-4, 3-5, 3-6, 3-7, 3-9, 3-10, 3-11, 3-12, 3-13, 3-14, 3-15, 3-16, 3-28, 3-29, 3-30, 4-1, 4-4, 4-9, 4-20, 4-21, 4-24, 4-25, 4-27, 5-1, 5-4, 5-5, 5-6, 5-7, 5-8, 5-11, 5-19, 5-20, 5-26, 7-12, E-17 |
| G | GROUP BY Clause 5-14, 5-15, 5-16, 5-18  
Group Functions i-5, 3-5, 5-1, 5-4, 5-5, 5-6, 5-11, 5-19, 5-20, 5-26, 7-12  
Group Functions in a Subquery 7-12 |
| H | HAVING Clause 5-22, 5-23, 5-24, 7-13 |
| I | Implicit Data Type Conversion 4-5, 4-6  
IN Operator 2-11  
Index 10-4, 11-23, 11-34, 11-37, 11-38, 11-39, 11-41, E-14  
INSERT Statement 9-6  
International Standards Organization (ISO) i-31  
INTERSECT Operator 8-19, 8-20  
INTERVAL YEAR TO MONTH 10-14 |
| J | Java i-9, i-35, i-56  
Joining Tables 6-6, C-7 |
| K | Keywords 1-8, 10-26, D-4 |
| L | LIKE Operator 2-12  
Literal 1-21, 1-22, 10-9 |
| M | MINUS Operator 8-22, 8-23  
MOD Function 3-19 |
| N | Naming 10-3, 10-5, 10-6, 10-11, 10-15, 10-31, 10-34, 10-37  
NEXTVAL 10-9, 10-27, 11-3, 11-22, 11-27, 11-28, 11-29, 11-33, 11-40  
NEXTVAL and CURRVAL Pseudocolumns 11-27, 11-28 |
Nonequijoins 6-3, 6-8, 6-19, 6-22, 6-23, 6-24, 6-25, 6-35, C-14, C-15, C-22

NOT NULL Constraint 10-20

NOT Operator 2-18

NULL Conditions 2-14

Null Value 1-14

Null Values 1-15, 1-20, 5-11, 7-21, 7-22, 9-8

NULLIF Function 4-31

Number Functions 3-16

NVL Function 4-28, 4-29

NVL2 Function 4-30

Object Relational i-16

OLTP i-11, i-16

ON clause 6-3, 6-5, 6-6, 6-8, 6-15, 6-16, 6-18, 6-19, 6-21, 6-22, 6-25, 6-31

ON DELETE CASCADE 10-26

ON DELETE SET NULL 10-26

OR Operator 2-17

Oracle Database 11g i-2, i-3, i-4, i-8, i-9, i-10, i-11, i-14, i-15, i-29, i-33, i-49, i-50, i-53, i-54, i-55, i-56, 3-24, 7-8, 10-14, 10-36, 10-38

Oracle Enterprise Manager Grid Control 10g i-13, i-56

Oracle Fusion Middleware i-12, i-13, i-56

Oracle Server 8-6

Oracle SQL Developer i-2, i-3, i-7, i-8, i-15, i-29, i-32, i-33, i-34, i-35, i-36, i-37, i-38, i-40, i-41, i-45, i-47, i-48, i-49, i-50, i-57, i-58, i-59, E-20

ORDBMS i-2, i-56

Order 2-40, 3-35, 4-25, 4-45, 6-39, B-2, C-26

ORDER BY Clause 2-23, 8-28

PRIMARY KEY Constraint 10-23

Projection 1-4
P
  Pseudocolumns 11-27, 11-28

Q
  q operator 1-23
  Queries i-5, 5-19, 5-20, 6-2, 7-1, 7-8, 8-4, 8-5, 10-27,
    C-2
  Query i-30, 7-15, D-3, E-11

R
  RDBMS i-2, i-18, i-25, i-27, i-56, 9-43
  Read Consistency 9-40, 9-41
  Read-only tables 10-3, 10-6, 10-11, 10-15, 10-31, 10-34, 10-37
  REFERENCES 10-25, 10-26, 10-28
  Relational Database i-16, i-18, i-19, i-27, i-28
  ROUND and TRUNC Functions 3-30
  ROUND Function 3-17
  RR Date Format 3-22, 4-22
  Rules of Precedence 1-12, 2-20, 2-21

S
  Schema i-6, i-51, 10-5, 10-8, 11-1, 11-35, B-2, E-4
  SELECT Statement i-5, 1-1, 1-5, 1-19, 1-28, 8-26, 9-43
  Selection 1-4, 2-4
  Sequences 11-23, 11-24, E-13
  Set operators 8-4, 8-5
  SET VERIFY ON 2-36
  Sorting i-5, 2-1, 2-3, 2-19, 2-22, 2-24, 2-25, 2-26, 2-34,
    2-38
  SQL Developer i-2, i-3, i-7, i-8, i-9, i-15, i-29, i-32,
    i-33, i-34, i-35, i-36, i-37, i-38, i-40, i-41, i-45, i-47, i-48,
    i-49, i-50, i-53, i-54, i-57, i-58, i-59, 1-6, 1-8, 1-9, 1-14,
    1-17, 1-26, 1-30, 2-28, 2-29, 2-30, 2-31, 2-33, 2-35, 2-36, 6-16,
    9-4, 9-21, 9-27, 9-31, 9-32, 9-43, 10-9, 10-40, 11-8, C-10, E-1,
    E-2, E-3, E-4, E-16, E-18, E-19, E-20, E-21
  Subquery 7-3, 7-4, 7-5, 7-6, 7-8, 7-9, 7-12, 7-15, 7-16,
    7-20, 7-21, 7-22, 9-17, 10-32, 10-33
  Substitution Variables 2-27, 2-28, 2-31
S
Synonym i-23, 10-4, 11-23, 11-34, 11-41, 11-42, 11-43, E-15
SYSDATE Function 3-24

T
TO_CHAR Function 4-11, 4-16, 4-17, 4-18, 4-19
Transactions 9-26, 9-27
TRUNC Function 3-18

U
UNION ALL Operator 8-16, 8-17
UNION Operator 8-13, 8-14, 8-15
UNIQUE Constraint 10-21, 10-22
Unique Identifier i-23
UPDATE Statement 9-15
USING Clause 6-11, 6-13, 6-14
Using Snippets E-16, E-17

V
VARIANCE 5-5, 5-8, 5-27
VERIFY Command 2-36
Views i-40, 11-6, 11-7, E-3, E-11, E-12

W
When to Create an Index 11-38
WHERE Clause 2-6, 6-10
WITH CHECK OPTION 11-8, 11-9, 11-17

X
XML i-9, i-14, i-37, i-38, i-56
Preface
Profile

Before You Begin This Course

Before you begin this course, you should be able to use a graphical user interface (GUI). The prerequisite is a familiarity with data processing concepts and techniques.

How This Course Is Organized

Oracle Database 11g: SQL Fundamentals I is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills that are introduced.
Related Publications

Oracle Publications

<table>
<thead>
<tr>
<th>Title</th>
<th>Part Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle® Database Reference 11g Release 1 (11.1)</td>
<td>B28320-01</td>
</tr>
<tr>
<td>Oracle® Database SQL Language Reference 11g Release 1 (11.1)</td>
<td>B28286-01</td>
</tr>
<tr>
<td>Oracle® Database Concepts 11g Release 1 (11.1)</td>
<td>B28318-01</td>
</tr>
<tr>
<td>Oracle® Database SQL Developer User's Guide Release 1.2</td>
<td>E10406-01</td>
</tr>
</tbody>
</table>

Additional Publications

- System release bulletins
- Installation and user’s guides
- read.me files
- International Oracle User’s Group (IOUG) articles
- Oracle Magazine
**Typographic Conventions**

What follows are two lists of typographical conventions that are used specifically within text or within code.

### Typographic Conventions Within Text

<table>
<thead>
<tr>
<th>Convention</th>
<th>Object or Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uppercase</td>
<td>Commands, functions, column names, table names, PL/SQL objects, schemas</td>
<td>Use the <code>SELECT</code> command to view information stored in the <code>LAST_NAME</code> column of the <code>EMPLOYEES</code> table.</td>
</tr>
<tr>
<td>Lowercase, italic</td>
<td>Filenames, syntax variables, usernames, passwords</td>
<td><em>where: role</em> is the name of the role to be created.</td>
</tr>
<tr>
<td>Initial cap</td>
<td>Trigger and button names</td>
<td>Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.</td>
</tr>
<tr>
<td>Italic</td>
<td>Books, names of courses and manuals, and emphasized words or phrases</td>
<td>For more information on the subject see <em>Oracle SQL Reference Manual</em></td>
</tr>
<tr>
<td>Quotation marks</td>
<td>Lesson module titles referenced within a course</td>
<td>Do <em>not</em> save changes to the database.</td>
</tr>
</tbody>
</table>

**Lesson Module:** This subject is covered in Lesson 3, “Working with Objects.”
## Typographic Conventions (continued)

### Typographic Conventions Within Code

<table>
<thead>
<tr>
<th>Convention</th>
<th>Object or Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uppercase</td>
<td>Commands, functions</td>
<td><code>SELECT employee_id</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>FROM employees;</code></td>
</tr>
<tr>
<td></td>
<td>Lowercase, italic</td>
<td><code>CREATE ROLE role;</code></td>
</tr>
<tr>
<td>Initial cap</td>
<td>Forms triggers</td>
<td><code>Form module: ORD</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Trigger level: S_ITEM.QUANTITY item</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Trigger name: When-Validate-Item</code></td>
</tr>
<tr>
<td></td>
<td>Lowercase</td>
<td><code>OG_ACTIVATE_LAYER</code></td>
</tr>
<tr>
<td></td>
<td>Column names, table names, filenames,</td>
<td><code>(OG_GET_LAYER ('prod_pie_layer'))</code></td>
</tr>
<tr>
<td></td>
<td>PL/SQL objects</td>
<td><code>SELECT last_name</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>FROM employees;</code></td>
</tr>
<tr>
<td>Bold</td>
<td>Text that must be entered by a user</td>
<td><code>CREATE USER scott</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>IDENTIFIED BY tiger;</code></td>
</tr>
</tbody>
</table>
Introduction
Lesson Objectives

After completing this lesson, you should be able to do the following:

• Understand the goals of the course
• List the features of Oracle Database 11g
• Discuss the theoretical and physical aspects of a relational database
• Describe Oracle server’s implementation of RDBMS and object relational database management system (ORDBMS)
• Identify the development environments that can be used for this course
• Describe and use the key features of Oracle SQL Developer
• Describe the database and schema used in this course

Objectives

In this lesson, you gain an understanding of the relational database management system (RDBMS) and the object relational database management system (ORDBMS). You are also introduced to Oracle SQL Developer and SQL*Plus as development environments used for executing SQL statements, and for formatting and reporting purposes.
Lesson Agenda

• Course objectives, agenda, and appendixes used in the course
  • Overview of Oracle Database 11g and related products
  • Overview of relational database management concepts and terminologies
  • Introduction to SQL and its development environments
  • Overview of Oracle SQL Developer
  • Oracle Database 11g documentation and additional resources
**Course Objectives**

After completing this course, you should be able to:

- Identify the major components of Oracle Database 11g
- Retrieve row and column data from tables with the `SELECT` statement
- Create reports of sorted and restricted data
- Employ SQL functions to generate and retrieve customized data
- Run complex queries to retrieve data from multiple tables
- Run data manipulation language (DML) statements to update data in Oracle Database 11g
- Run data definition language (DDL) statements to create and manage schema objects

**Course Objectives**

This course offers you an introduction to Oracle Database 11g database technology. In this class, you learn the basic concepts of relational databases and the powerful SQL programming language. This course provides the essential SQL skills that enable you to write queries against single and multiple tables, manipulate data in tables, create database objects, and query metadata.
Course Agenda

• Day 1:
  – Introduction
  – Retrieving Data Using the SQL **SELECT** Statement
  – Restricting and Sorting Data
  – Using Single-Row Functions to Customize Output
  – Using Conversion Functions and Conditional Expressions

• Day 2:
  – Reporting Aggregated Data Using the Group Functions
  – Displaying Data from Multiple Tables
  – Using Subqueries to Solve Queries
  – Using the Set Operators
Course Agenda

• Day 3:
  – Manipulating Data
  – Using DDL Statements to Create and Manage Tables
  – Creating Other Schema Objects
Appendixes Used in the Course

- Appendix A: Practice Solutions
- Appendix B: Table Descriptions
- Appendix C: Oracle Join Syntax
- Appendix D: Using SQL*Plus
- Appendix E: Performing DML and DDL Operations Using the Oracle SQL Developer GUI
- Additional Practices
- Additional Practices Solutions
Lesson Agenda

- Course objectives, course agenda, and appendixes used in this course
- Overview of Oracle Database 11g and related products
  - Overview of relational database management concepts and terminologies
  - Introduction to SQL and its development environments
  - Overview of Oracle SQL Developer
  - The HR schema and the tables used in this course
  - Oracle Database 11g documentation and additional resources
Oracle Database 11g: Focus Areas

Oracle Database 11g offers extensive features across the following focus areas:

- **Infrastructure Grids**: The Infrastructure Grid technology of Oracle enables pooling of low-cost servers and storage to form systems that deliver the highest quality of service in terms of manageability, high availability, and performance. Oracle Database 11g consolidates and extends the benefits of grid computing. Apart from taking full advantage of grid computing, Oracle Database 11g has unique change assurance features to manage changes in a controlled and cost effective manner.

- **Information Management**: Oracle Database 11g extends the existing information management capabilities in content management, information integration, and information life cycle management areas. Oracle provides content management of advanced data types such as Extensible Markup Language (XML), text, spatial, multimedia, medical imaging, and semantic technologies.

- **Application Development**: Oracle Database 11g has capabilities to use and manage all the major application development environments such as PL/SQL, Java/JDBC, .NET and Windows, PHP, SQL Developer, and Application Express.
Oracle Database 11g

Organizations need to support multiple terabytes of information for users who demand fast and secure access to business applications round-the-clock. The database systems must be reliable and must be able to recover quickly in the event of any kind of failure. Oracle Database 11g is designed along the following feature areas to help organizations manage infrastructure grids easily and deliver high-quality service:

- **Manageability**: By using some of the change assurance, management automation, and fault diagnostics features, the database administrators (DBAs) can increase their productivity, reduce costs, minimize errors, and maximize quality of service. Some of the useful features that promote better management are Database Replay facility, the SQL Performance Analyzer, and the Automatic SQL Tuning facility.

- **High availability**: By using the high availability features, you can reduce the risk of down time and data loss. These features improves online operations and enable faster database upgrades.
Oracle Database 11g (continued)

- **Performance:** By using capabilities such as SecureFiles, compression for online transaction processing (OLTP), Real Application Clusters (RAC) optimizations, Result Caches and so on, you can greatly improve the performance of your database. Oracle Database 11g enables organizations to manage large, scalable transactional and data warehousing systems that deliver fast data access using low-cost modular storage.

- **Security:** Oracle Database 11g helps organizations protect their information with unique secure configurations, data encryption and masking, and sophisticated auditing capabilities. It delivers a secure and scalable platform for reliable and fast access to all types of information by using the industry-standard interfaces.

- **Information integration:** Oracle Database 11g has many features to better integrate data throughout the enterprise. It also supports advanced information life cycle management capabilities. This helps you manage the changing data in your database.
Oracle Fusion Middleware

Portfolio of leading, standards-based, and customer-proven software products that spans a range of tools and services from J2EE and developer tools, through integration services, business intelligence, collaboration, and content management.

Oracle Fusion Middleware

Oracle Fusion Middleware is a comprehensive and well-integrated family of products that offers complete support for development, deployment, and management of Service-Oriented Architecture (SOA). SOA facilitates the development of modular business services that can be easily integrated and reused, thereby reducing development and maintenance costs, and providing higher quality of services. Oracle Fusion Middleware’s pluggable architecture enables you to leverage your investments in any existing application, system, or technology. Its unbreakable core technology minimizes the disruption caused by planned or unplanned outages.

Some of the products from the Oracle Fusion Middleware family include:

- **Enterprise Application Server**: Application Server
- **Integration and Process Management**: BPEL Process Manager, Oracle Business Process Analysis Suite
- **Development Tools**: Oracle Application Development Framework, JDeveloper, SOA Suite
- **Business Intelligence**: Oracle Business Activity Monitoring, Oracle Data Integrator
- **Systems Management**: Enterprise Manager
- **Identity Management**: Oracle Identity Management
- **Content Management**: Oracle Content Database Suite
- **User Interaction**: Portal, WebCenter
Oracle Enterprise Manager Grid Control 10g

- Efficient Oracle Fusion Middleware management
- Simplifying application and infrastructure life cycle management
- Improved database administration and application management capabilities

Spanning applications, middleware, and database management, Oracle Enterprise Manager Grid Control 10g delivers integrated enterprise management for Oracle and non-Oracle systems.

Oracle Enterprise Manager Grid Control 10g features advanced Oracle Fusion Middleware management capabilities for the services that business applications rely upon, including SOA, Business Activity Monitoring, and Identity Management.

- **Wide-ranging management functionality** for your applications including service-level management, application performance management, configuration management, and change automation
- **Built-in grid automation capabilities** means that information technology responds proactively to fluctuating demand and implements new services more quickly so that businesses can thrive.
- **In-depth diagnostics and readily available remediation** across a range of applications including custom-built applications, Oracle E-Business Suite, PeopleSoft, Siebel, Oracle Fusion Middleware, Oracle Database, and underlying infrastructure
- **Extensive life cycle management capabilities** extends grid computing by providing solutions for the entire application and infrastructure life cycle, including test, stage, and production through operations. It has simplified patch management with synchronized patching, additional operating system support, and conflict detection features.
Oracle BI Publisher

- Provides a central architecture for authoring, managing, and delivering information in secure and multiple formats
- Reduces complexity and time to develop, test, and deploy all kinds of reports
  - Financial Reports, Invoices, Sales or Purchase orders, XML, and EDI/EFT(eText documents)
- Enables flexible customizations
  - For example, a Microsoft Word document report can be generated in multiple formats such as PDF, HTML, Excel, RTF, and so on.

Oracle BI Publisher

Oracle Database 11g also includes Oracle BI Publisher—the enterprise reporting solution from Oracle. Oracle BI Publisher (formerly known as XML Publisher) offers the most efficient and scalable reporting solution available for complex, distributed environments.

Oracle BI Publisher reduces the high costs associated with the development, customization, and maintenance of business documents, while increasing the efficiency of reports management. By using a set of familiar desktop tools, users can create and maintain their own report formats based on data queries created by the IT staff or developers.

Oracle BI Publisher report formats can be designed using Microsoft Word or Adobe Acrobat—tools that most users are already familiar with. Oracle BI Publisher also enables you to bring in data from multiple data sources into a single output document. You can deliver reports via printer, email, or fax. You can publish your report to a portal. You can even allow users to collaboratively edit and manage reports on the Web-based Distributed Authoring and Versioning (WebDav) Web servers.
Lesson Agenda

• Course objectives, course agenda, and appendixes used in this course
• Overview of Oracle Database 11g and related products
• Overview of relational database management concepts and terminologies
• Introduction to SQL and its development environments
• Overview of Oracle SQL Developer
• The HR schema and the tables used in this course
• Oracle Database 11g documentation and additional resources
Relational and Object Relational Database Management Systems

- Relational model and object relational model
- User-defined data types and objects
- Fully compatible with relational database
- Supports multimedia and large objects
- High-quality database server features

The Oracle server supports both the relational and the object relational database models.

The Oracle server extends the data-modeling capabilities to support an object relational database model that provides object-oriented programming, complex data types, complex business objects, and full compatibility with the relational world.

It includes several features for improved performance and functionality of the OLTP applications, such as better sharing of run-time data structures, larger buffer caches, and deferrable constraints. Data warehouse applications benefit from enhancements such as parallel execution of insert, update, and delete operations; partitioning; and parallel-aware query optimization. Operating within the Network Computing Architecture (NCA) framework, the Oracle model supports client/server and Web-based applications that are distributed and multitiered.

For more information about the relational and object relational model, see the Oracle Database Concepts 11g Release 1 (11.1) manual.
Data Storage on Different Media

Every organization has some information needs. A library keeps a list of members, books, due dates, and fines. A company needs to save information about its employees, departments, and salaries. These pieces of information are called *data*.

Organizations can store data in various media and in different formats, such as a hard copy document in a filing cabinet, or data stored in electronic spreadsheets, or in databases.

A *database* is an organized collection of information.

To manage databases, you need a database management system (DBMS). A DBMS is a program that stores, retrieves, and modifies data in databases on request. There are four main types of databases: *hierarchical, network, relational*, and (most recently) *object relational*. 
Relational Database Concept

• Dr. E. F. Codd proposed the relational model for database systems in 1970.
• It is the basis for the relational database management system (RDBMS).
• The relational model consists of the following:
  – Collection of objects or relations
  – Set of operators to act on the relations
  – Data integrity for accuracy and consistency

The principles of the relational model were first outlined by Dr. E. F. Codd in a June 1970 paper titled “A Relational Model of Data for Large Shared Data Banks.” In this paper, Dr. Codd proposed the relational model for database systems.

The common models used at that time were hierarchical and network, or even simple flat-file data structures. Relational database management systems (RDBMS) soon became very popular, especially for their ease of use and flexibility in structure. In addition, a number of innovative vendors, such as Oracle, supplemented the RDBMS with a suite of powerful, application development and user-interface products, thereby providing a total solution.

Components of the Relational Model
• Collections of objects or relations that store the data
• A set of operators that can act on the relations to produce other relations
• Data integrity for accuracy and consistency

For more information, see An Introduction to Database Systems, Eighth Edition (Addison-Wesley: 2004), written by Chris Date.
Definition of a Relational Database

A relational database is a collection of relations or two-dimensional tables.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Koohfar</td>
<td>NKOOHFAR</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
<td>LDEHAAN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>201</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
</tr>
</tbody>
</table>

Definition of a Relational Database

A relational database uses relations or two-dimensional tables to store information. For example, you might want to store information about all the employees in your company. In a relational database, you create several tables to store different pieces of information about your employees, such as an employee table, a department table, and a salary table.
Data Models

Models are the cornerstone of design. Engineers build a model of a car to work out any details before putting it into production. In the same manner, system designers develop models to explore ideas and improve the understanding of database design.

Purpose of Models

Models help communicate the concepts that are in people’s minds. They can be used to do the following:

- Communicate
- Categorize
- Describe
- Specify
- Investigate
- Evolve
- Analyze
- Imitate

The objective is to produce a model that fits a multitude of these uses, can be understood by an end user, and contains sufficient detail for a developer to build a database system.
Entity Relationship Model

- Create an entity relationship diagram from business specifications or narratives:

```
EMPLOYEE
#* number
* name
0 job title

DEPARTMENT
#* number
* name
0 location
```

- Scenario:
  - “. . . Assign one or more employees to a department . . .”
  - “. . . Some departments do not yet have assigned employees . . .”

Entity Relationship Model

In an effective system, data is divided into discrete categories or entities. An entity relationship (ER) model is an illustration of the various entities in a business and the relationships among them. An ER model is derived from business specifications or narratives and built during the analysis phase of the system development life cycle. ER models separate the information required by a business from the activities performed within the business. Although businesses can change their activities, the type of information tends to remain constant. Therefore, the data structures also tend to be constant.
Entity Relationship Model (continued)

Benefits of ER Modeling:
• Documents information for the organization in a clear, precise format
• Provides a clear picture of the scope of the information requirement
• Provides an easily understood pictorial map for database design
• Offers an effective framework for integrating multiple applications

Key Components
• Entity: An aspect of significance about which information must be known. Examples are departments, employees, and orders.
• Attribute: Something that describes or qualifies an entity. For example, for the employee entity, the attributes would be the employee number, name, job title, hire date, department number, and so on. Each of the attributes is either required or optional. This state is called optionality.
• Relationship: A named association between entities showing optionality and degree. Examples are employees and departments, and orders and items
Entity Relationship Modeling Conventions

Entity:
- Singular, unique name
- Uppercase
- Soft box
- Synonym in parentheses

Attribute:
- Singular name
- Lowercase
- Mandatory marked with “*”
- Optional marked with “o”

Unique Identifier (UID)
Primary marked with “#”
Secondary marked with “(#)”

ER Modeling Conventions

Entities
To represent an entity in a model, use the following conventions:
- Singular, unique entity name
- Entity name in uppercase
- Soft box
- Optional synonym names in uppercase within parentheses: ( )

Attributes
To represent an attribute in a model, use the following conventions:
- Singular name in lowercase
- Asterisk (*) tag for mandatory attributes (that is, values that must be known)
- Letter “o” tag for optional attributes (that is, values that may be known)

Relationships

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dashed line</td>
<td>Optional element indicating “maybe”</td>
</tr>
<tr>
<td>Solid line</td>
<td>Mandatory element indicating “must be”</td>
</tr>
<tr>
<td>Crow’s foot</td>
<td>Degree element indicating “one or more”</td>
</tr>
<tr>
<td>Single line</td>
<td>Degree element indicating “one and only one”</td>
</tr>
</tbody>
</table>
ER Modeling Conventions (continued)

Relationships
Each direction of the relationship contains:

• **A label:** for example, *taught by* or *assigned to*
• **An optionality:** either *must be* or *maybe*
• **A degree:** either *one and only one* or *one or more*

**Note:** The term *cardinality* is a synonym for the term *degree*.

Each source entity {may be | must be} in relation {one and only one | one or more} with the destination entity.

**Note:** The convention is to read clockwise.

Unique Identifiers
A unique identifier (UID) is any combination of attributes or relationships, or both, that serves to distinguish occurrences of an entity. Each entity occurrence must be uniquely identifiable.

• Tag each attribute that is part of the UID with a hash sign (#).
• Tag secondary UIDs with a hash sign in parentheses (#).
Relating Multiple Tables

- Each row of data in a table is uniquely identified by a primary key.
- You can logically relate data from multiple tables using foreign keys.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Koontz</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Hean</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hundal</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
<td>50</td>
</tr>
<tr>
<td>107</td>
<td>Elena</td>
<td>Lorenz</td>
<td>60</td>
</tr>
</tbody>
</table>

Table name: EMPLOYEES

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>80</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>10</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
<tr>
<td>50</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>100</td>
<td>Accounting</td>
<td>205</td>
<td>1700</td>
</tr>
<tr>
<td>150</td>
<td>Contracting</td>
<td>(null)</td>
<td>1700</td>
</tr>
</tbody>
</table>

Table name: DEPARTMENTS

Relating Multiple Tables

Each table contains data that describes exactly one entity. For example, the EMPLOYEES table contains information about employees. Categories of data are listed across the top of each table, and individual cases are listed below. By using a table format, you can readily visualize, understand, and use information.

Because data about different entities is stored in different tables, you may need to combine two or more tables to answer a particular question. For example, you may want to know the location of the department where an employee works. In this scenario, you need information from the EMPLOYEES table (which contains data about employees) and the DEPARTMENTS table (which contains information about departments). With an RDBMS, you can relate the data in one table to the data in another by using the foreign keys. A foreign key is a column (or a set of columns) that refers to a primary key in the same table or another table.

You can use the ability to relate data in one table to data in another to organize information in separate, manageable units. Employee data can be kept logically distinct from the department data by storing it in a separate table.
Guidelines for Primary Keys and Foreign Keys

• You cannot use duplicate values in a primary key.
• Primary keys generally cannot be changed.
• Foreign keys are based on data values and are purely logical (not physical) pointers.
• A foreign key value must match an existing primary key value or unique key value, otherwise it must be null.
• A foreign key must reference either a primary key or a unique key column.
Relational Database Terminology

A relational database can contain one or many tables. A table is the basic storage structure of an RDBMS. A table holds all the data necessary about something in the real world, such as employees, invoices, or customers.

The slide shows the contents of the EMPLOYEES table or relation. The numbers indicate the following:

1. A single row (or tuple) representing all the data required for a particular employee. Each row in a table should be identified by a primary key, which permits no duplicate rows. The order of rows is insignificant; specify the row order when the data is retrieved.

2. A column or attribute containing the employee number. The employee number identifies a unique employee in the EMPLOYEES table. In this example, the employee number column is designated as the primary key. A primary key must contain a value and the value must be unique.

3. A column that is not a key value. A column represents one kind of data in a table; in this example, the data is the salaries of all the employees. Column order is insignificant when storing data; specify the column order when the data is retrieved.
Relational Database Terminology (continued)

4. A column containing the department number, which is also a foreign key. A foreign key is a column that defines how tables relate to each other. A foreign key refers to a primary key or a unique key in the same table or in another table. In the example, DEPARTMENT_ID uniquely identifies a department in the DEPARTMENTS table.

5. A field can be found at the intersection of a row and a column. There can be only one value in it.

6. A field may have no value in it. This is called a null value. In the EMPLOYEES table, only those employees who have the role of sales representative have a value in the COMMISSION_PCT (commission) field.
Lesson Agenda

• Course objectives, course agenda, and appendixes used in this course
• Overview of Oracle Database 11g and related products
• Overview of relational database management concepts and terminologies
• Introduction to SQL and its development environments
• Overview of Oracle SQL Developer
• The HR schema and the tables used in this course
• Oracle Database 11g documentation and additional resources
Using SQL to Query Your Database

Structured query language (SQL) is:

- The ANSI standard language for operating relational databases
- Efficient, easy to learn, and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)

```
SELECT department_name
FROM departments;
```

Using SQL to Query Your Database

In a relational database, you do not specify the access route to the tables, and you do not need to know how the data is arranged physically.

To access the database, you execute a structured query language (SQL) statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. SQL is a set of statements with which all programs and users access data in an Oracle database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications, in turn, must use SQL when executing the user’s request.

SQL provides statements for a variety of tasks, including:
- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the preceding tasks in one consistent language and enables you to work with data at a logical level.
SQL Statements

SQL statements supported by Oracle comply with industry standards. Oracle Corporation ensures future compliance with evolving standards by actively involving key personnel in SQL standards committees. The industry-accepted committees are ANSI and International Standards Organization (ISO). Both ANSI and ISO have accepted SQL as the standard language for relational databases.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT, INSERT, UPDATE, DELETE, MERGE</td>
<td>Retrieves data from the database, enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as <em>data manipulation language</em> (DML)</td>
</tr>
<tr>
<td>CREATE, ALTER, DROP, RENAME, TRUNCATE, COMMENT</td>
<td>Sets up, changes, and removes data structures from tables. Collectively known as <em>data definition language</em> (DDL)</td>
</tr>
<tr>
<td>GRANT, REVOKE</td>
<td>Provides or removes access rights to both the Oracle database and the structures within it.</td>
</tr>
<tr>
<td>COMMIT, ROLLBACK, SAVEPOINT</td>
<td>Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions</td>
</tr>
</tbody>
</table>
Development Environments for SQL

In this course:

- Primarily use Oracle SQL Developer, Release 1.2
- Use SQL*Plus:
  - In case you do not have access to Oracle SQL Developer
  - Or when any command does not work in Oracle SQL Developer

Development Environments for SQL

This course has been developed using Oracle SQL Developer as the tool for running the SQL statements discussed in the examples in the slide and the practices. For commands that are not supported by Oracle SQL Developer, use the SQL*Plus environment.
Lesson Agenda

• Course objectives, course agenda, and appendixes used in this course
• Overview of Oracle Database 11g and related products
• Overview of relational database management concepts and terminologies
• Introduction to SQL and its development environments
• **Overview of Oracle SQL Developer**
• The HR schema and the tables used in this course
• Oracle Database 11g documentation and additional resources
What Is Oracle SQL Developer?

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

Oracle SQL Developer, the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using the standard Oracle database authentication. When connected, you can perform operations on objects in the database.

**Note:** The Oracle SQL Developer, Release 1.2 is called the *Migration release* because it tightly integrates with *Developer Migration Workbench*. Therefore, it provides users with a single point to browse database objects and data in third-party databases, and to migrate from these databases to Oracle. You can also connect to schemas for selected third-party (non-Oracle) databases, such as MySQL, Microsoft SQL Server, and Microsoft Access, and view metadata and data in these databases.

Additionally, Oracle SQL Developer, Release 1.2 includes support for Oracle Application Express, Release 3.0.1 (Oracle APEX).
Specifications of Oracle SQL Developer

- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
  - Unzip the downloaded Oracle SQL Developer kit and double-click sqldeveloper.exe to start Oracle SQL Developer.
- Connects to any Oracle Database, version 9.2.0.1 and later
- Freely downloadable from the following link:
- Needs JDK 1.5 installed on your system, which can be downloaded from the following link:

Oracle SQL Developer is developed in Java leveraging the Oracle JDeveloper integrated development environment (IDE). It is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms. You can install Oracle SQL Developer on the database server and connect remotely from your desktop, thus avoiding client/server network traffic.

Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, so Oracle Home is not required. Oracle SQL Developer does not require an installer and you need to simply unzip the downloaded file. With Oracle SQL Developer, users can connect to the Oracle Database, version 9.2.0.1 and later, and all Oracle database editions including the Express Edition.

Oracle SQL Developer can be downloaded with the following packaging options:
- Oracle SQL Developer for Windows (option to download with or without Java Development Kit (JDK) 1.5)
- Oracle SQL Developer for Multiple Platforms (you should have JDK 1.5 already installed)
- Oracle SQL Developer for Mac OS X platforms (you should have JDK 1.5 already installed)
- Oracle SQL Developer RPM for Linux (you should have JDK 1.5 already installed)

Note that Oracle SQL Developer, Release 1.2 is also certified with JDK 6.0.
Oracle SQL Developer Interface

Oracle SQL Developer has two main navigation tabs:

- **Connections tab**: By using this tab, you can browse database objects and users to which you have access.
- **Reports tab**: By using this tab, you can run predefined reports, or create and add your own reports.

Oracle SQL Developer uses the left pane for navigation to find and select objects, and the right pane to display information about selected objects. You can customize many aspects of the appearance and behavior of Oracle SQL Developer by setting preferences. The menus at the top contain standard entries, plus entries for features specific to Oracle SQL Developer.

1. **View**: Contains options that affect what is displayed in the Oracle SQL Developer interface
2. **Navigate**: Contains options for navigating to panes and in the execution of subprograms
3. **Run**: Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected
4. **Debug**: Contains options relevant when a function or procedure is selected for debugging
5. **Source**: Contains options for use when editing functions and procedures
6. **Migration**: Contains options related to migrating third-party databases to Oracle
7. **Tools**: Invokes tools such as SQL*Plus, Preferences, and SQL Worksheet

**Note**: You must define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures/functions.
Creating a Database Connection

- You must have at least one database connection to use Oracle SQL Developer.
- You can create and test connections for:
  - Multiple databases
  - Multiple schemas
- Oracle SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an XML file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

Creating a Database Connection

A connection is an Oracle SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use Oracle SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory. But, it can also be in the directory specified by the `TNS_ADMIN` environment variable or the registry value. When you start Oracle SQL Developer and display the Database Connections dialog box, Oracle SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows systems, if the `tnsnames.ora` file exists but Oracle SQL Developer is not using its connections, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it later.

You can create additional connections as different users to the same database or to connect to the different databases.
Creating a Database Connection

Creating a Database Connection (continued)

To create a database connection, perform the following steps:
1. On the Connections tabbed page, right-click Connections and select New Connection.
2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.
   1. From the Role drop-down list, you can select either default or SYSDBA (you will select SYSDBA for the sys user or any user with DBA privileges).
   2. You can select the connection type as:
      - Basic: In this type, you enter the host name and system identifier (SID) for the database that you want to connect to. The Port is already set to 1521. Or, you can also enter the Service name directly if you are using a remote database connection.
      - TNS: You select any one of the database aliases imported from the tnsnames.ora file
      - Advanced: You define a custom JDBC URL to connect to the database.
3. Click Test to make sure that the connection has been set correctly.
4. Click Connect.

If you select the Save Password check box, the password is saved to an XML file. So, after you close the Oracle SQL Developer connection and open it again, you will not be prompted for the password.
Creating a Database Connection (continued)

3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions, for example, dependencies, details, statistics, and so on.

**Note:** From the same New/Select Database Connection window, you can define connections to non-Oracle data sources by using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.
Browsing Database Objects

Use the Connections Navigator to:

• Browse through many objects in a database schema
• Review the definitions of objects at a glance

After you have created a database connection, you can use the Connections Navigator to browse through many objects in a database schema including Tables, Views, Indexes, Packages, Procedures, Triggers, Types, and so on.

Oracle SQL Developer uses the left pane for navigation to find and select objects and the right pane to display information about the selected objects. You can customize many aspects of the appearance of Oracle SQL Developer by setting preferences.

You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed in an easy-to-read tabbed page.

If you want to see the definition of the EMPLOYEES table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click EMPLOYEES. By default, the Columns tab is selected. It shows the column description of the table. By using the Data tab, you can view the tables data and also enter new rows, update data, and commit these changes to the database.
Using the SQL Worksheet

• Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
• Specify any actions that can be processed by the database connection associated with the Worksheet.

Using the SQL Worksheet

When you connect to a database, a SQL Worksheet window for that connection is automatically opened. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL Worksheet to the Oracle database. However, the SQL*Plus commands used in Oracle SQL Developer have to be interpreted by the SQL Worksheet before being passed to the database.

The SQL Worksheet currently supports a number of SQL*Plus commands. Those commands that are not supported by the SQL Worksheet are ignored and not sent to the Oracle database. Through the SQL Worksheet, you can execute SQL statements and some of the SQL*Plus commands.

You can display a SQL Worksheet by using any of the following two options:
• Select Tools > SQL Worksheet.
• Click the Open SQL Worksheet icon available on the main toolbar.
Using the SQL Worksheet (continued)

You may want to use shortcut keys or icons to perform certain tasks, such as executing a SQL statement, running a script, or viewing the history of the SQL statements that you have executed.

You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement**: This executes the statement at the cursor in the Enter SQL Statement box. Alternatively, you can press [F9]. The output is generally shown in a formatted manner in the Results tab page.

2. **Run Script**: This executes all statements in the Enter SQL Statement box using the Script Runner. The output is generally shown in the conventional script format in the Scripts tab page.

3. **Commit**: This writes any changes to the database and ends the transaction.

4. **Rollback**: This discards any changes to the database, without writing them to the database, and ends the transaction.

5. **Cancel**: This stops the execution of any statements currently being executed.
Using the SQL Worksheet (continued)

6. **SQL History**: This displays a dialog box with information about the SQL statements that you have executed.

7. **Execute Explain Plan**: This generates the execution plan, which you can see by clicking the Explain tab.

8. **Autotrace**: This displays trace-related information when you execute the SQL statement by clicking the Autotrace icon. This information can help you to identify the SQL statements that will benefit from tuning.

9. **Clear**: This erases the statement or statements in the Enter SQL Statement box. Alternatively, press and hold [Ctrl] + [D] to erase the statements.
Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

In the SQL Worksheet, you can use the Enter SQL Statement box to enter a single or multiple SQL statements. For a single statement, the semicolon at the end is optional. When you enter the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the Execute Statement icon. Alternatively, you can press [F9].

To execute multiple SQL statements and see the results, click the Run Script icon. Alternatively, you can press [F5].

The example in the slide shows the difference in output for the same query when F9 key or Execute Statement is used versus the output when F5 or Run Script is used.
Formatting the SQL Code

You may want to beautify the indentation, spacing, capitalization, and the line separation of the SQL code. Oracle SQL Developer has the feature for formatting the SQL code.

To format the SQL code, right-click in the statement area, and select Format SQL.

In the example in the slide, before formatting, the keywords are not capitalized and the statement is not properly indented in the SQL code. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.
Saving SQL Statements

In most of the practices that you will perform, you will need to save a particular query in the SQL Worksheet as a .sql file. To do so, perform the following:

1. From the File menu, select Save or Save As (if you are renaming a current .sql script) or you can right-click the SQL Worksheet and select Save File. Alternatively, you can press and hold [CTRL] + [S].
2. In the Save dialog box, enter the appropriate filename. Make sure the extension is .sql or the File type is selected as SQL Script (*.sql). Click Save.
3. The SQL Worksheet is renamed to the filename that you saved the script as. Make sure you do not enter any other SQL statements in the same worksheet. To continue with other SQL queries, open a new worksheet.

Note: For this course, you need to save your sql scripts in the D:\labs\sql1\labs folder.
Running Script Files

To run the saved .sql script files, perform the following:

1. Right-click the SQL Worksheet and select Open File, or select Open from the File menu. Alternatively, you can press and hold [CTRL] + [O].
2. In the Open dialog box, move to the D:\labs\sql\labs folder, or to the location in which you saved the script file, select the file and click Open.
3. The script file opens in a new worksheet. Now, you can run the script by either clicking the Execute Statement icon or the Run Script icon. Again, make sure you do not enter any other SQL statements in the same worksheet. To continue with other SQL queries, open a new worksheet.

Note: You may want to set the default directory to D:\labs\sql1 folder, so that every time you try to open or save a script, SQL Developer chooses the same path to look for scripts. From Tools menu, select Preferences. In the Preferences dialog box, expand Database and select Worksheet Parameters. In the right pane, click Browse to set the default path to look for scripts and click OK.

Note: For more details on how to use the Oracle SQL Developer GUI interface for other data objects creation and data retrieval tasks, refer to Appendix G “Performing DML and DDL Operations Using the Oracle SQL Developer GUI.”
Starting SQL*Plus from Oracle SQL Developer

You can invoke the SQL*Plus command-line interface from Oracle SQL Developer.

Provide the location of the sqlplus.exe file only for the first time you invoke SQL*Plus.

Starting SQL*Plus from Oracle SQL Developer

The SQL Worksheet supports most of the SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL Worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database. To display the SQL*Plus command window, from the Tools menu, select SQL*Plus. To use this feature, the system on which you are using Oracle SQL Developer must have an Oracle Home directory or folder, with a SQL*Plus executable in that location. If the location of the SQL*Plus executable is not already stored in your Oracle SQL Developer preferences, you are asked to specify its location.

Note: If the Tools > SQL*Plus menu option is disabled, click the database connection, such as myconnection, in the Connections Navigator. The menu option is disabled if any SQL Worksheet is active.

For example, some of the SQL*Plus statements that are not supported by SQL Worksheet are:

- append
- archive
- attribute
- break

For a complete list of the SQL*Plus statements that are supported and not supported by the SQL Worksheet, refer to the SQL*Plus Statements Supported and Not Supported in SQL Worksheet topic in the Oracle SQL Developer online Help.
SQL Statements in SQL*Plus

In Oracle Database 11g, SQL*Plus is a command-line interface.

Oracle SQL*Plus is a command-line interface with which you can submit SQL statements and PL/SQL blocks for execution and receive the results in an application or command window.

SQL*Plus is:

- Shipped with the database
- Installed on a client and on the database server system
- Accessed through from an icon or the command line

Note: If you do not have access to Oracle SQL Developer and would prefer to use SQL*Plus, the classroom setup provides SQL*Plus icon on your desktop. It may also be useful to use it in cases where Oracle SQL Developer does not support any SQL* Plus command.
Lesson Agenda

- Course objectives, course agenda, and appendixes used in this course
- Overview of Oracle Database 11g and related products
- Overview of relational database management concepts and terminologies
- Introduction to SQL and its development environments
- Overview of Oracle SQL Developer
- The HR schema and the tables used in this course
- Oracle Database 11g documentation and additional resources
The Human Resources (HR) Schema Description

The Human Resources (HR) schema is a part of the Oracle Sample Schemas that can be installed in an Oracle database. The practice sessions in this course use data from the HR schema.

**Table Descriptions**

- **REGIONS** contains rows that represent a region such as America, Asia, and so on.
- **COUNTRIES** contains rows for countries, each of which is associated with a region.
- **LOCATIONS** contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
- **DEPARTMENTS** shows details about the departments in which the employees work. Each department may have a relationship representing the department manager in the EMPLOYEES table.
- **EMPLOYEES** contains details about each employee working for a department. Some employees may not be assigned to any department.
- **JOBS** contains the job types that can be held by each employee.
- **JOB_HISTORY** contains the job history of the employees. If an employee changes departments within a job or changes jobs within a department, then a new row is inserted into this table with the earlier job information of the employee.
Tables Used in the Course

The following main tables are used in this course:

- **EMPLOYEES** table: Gives details of all the employees
- **DEPARTMENTS** table: Gives details of all the departments
- **JOB_GRADES** table: Gives details of salaries for various grades

Apart from these tables, you will also use the other tables listed in the previous slide such as the **LOCATIONS** and the **JOB_HISTORY** table.

**Note:** The structure and data for all the tables are provided in Appendix B.
Lesson Agenda

• Course objectives, course agenda, and appendixes used in this course
• Overview of Oracle Database 11g and related products
• Overview of relational database management concepts and terminologies
• Introduction to SQL and its development environments
• Overview of Oracle SQL Developer
• The HR schema and the tables used in this course
• Oracle Database 11g documentation and additional resources
Oracle Database 11g Documentation

- Oracle Database New Features Guide 11g, Release 1 (11.1)
- Oracle Database Reference 11g, Release 1 (11.1)
- Oracle Database SQL Language Reference 11g, Release 1 (11.1)
- Oracle Database Concepts 11g, Release 1 (11.1)
- Oracle Database SQL Developer User's Guide, Release 1.2

Navigate to http://www.oracle.com/pls/db111/homepage to access the Oracle Database 11g documentation library.
Additional Resources

For additional information about the Oracle Database 11g, refer to the following:

- *Oracle Database 11g: New Features eStudies*
- *Oracle by Example series (OBE): Oracle Database 11g*
Summary

- Oracle Database 11g extends:
  - the benefits of infrastructure grids
  - the existing information management capabilities
  - the capabilities to use the major application development environments such as PL/SQL, Java/JDBC, .NET, XML, and so on
- The database is based on ORDBMS.
- Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.
- With the Oracle server, you can store and manage information by using SQL.

Summary

Relational database management systems are composed of objects or relations. They are managed by operations and governed by data integrity constraints.

Oracle Corporation produces products and services to meet your RDBMS needs. The main products are the following:
- Oracle Database 11g with which you store and manage information by using SQL
- Oracle Fusion Middleware with which you develop, deploy, and manage modular business services that can be integrated and reused
- Oracle Enterprise Manager Grid Control 10g, which you use to manage and automate administrative tasks across sets of systems in a grid environment

SQL

The Oracle server supports ANSI-standard SQL and contains extensions. SQL is the language that is used to communicate with the server to access, manipulate, and control data.
Practice I: Overview

This practice covers the following topics:

- Running the Oracle SQL Developer demo
- Starting Oracle SQL Developer, creating a new database connection, and browsing the HR tables

Practice I: Overview

In this practice, you perform the following:

- Run through the Oracle SQL Developer demo.
- Use Oracle SQL Developer to examine data objects in the ORA account assigned to you. The ORA accounts contain the HR schema tables.

Note the following location for the lab files:

`D:\labs\SQL1\labs`

If you are asked to save any lab files, save them in this location.

In any practice, there maybe exercises that are prefaced with the phrases “If you have time” or “If you want an extra challenge.” Work on these exercises only if you have completed all other exercises within the allocated time and would like a further challenge to your skills.

Perform the practices slowly and precisely. You can experiment with saving and running command files. If you have any questions at any time, ask your instructor.

Note: All written practices use Oracle SQL Developer as the development environment. Although it is recommended that you use Oracle SQL Developer, you can also use SQL*Plus that is available in this course.
Practice I

This is the first of many practices in this course. The solutions (if you require them) can be found in Appendix A. Practices are intended to cover most of the topics that are presented in the corresponding lesson.

Run Through the Oracle SQL Developer Demo: Creating a Database Connection

1. Access the demo “Creating a database connection” at:

Starting Oracle SQL Developer

2. Start Oracle SQL Developer using the sqldeveloper desktop icon.
   
   Note: When you start SQL Developer for the first time, you need to provide the path to the java.exe file. This is already done for you as a part of the classroom setup. In any case, if you are prompted, enter the following path:
   D:\app\Administrator\product\11.1.0\client_1\jdevstudio\jdk\bin

Creating a New Oracle SQL Developer Database Connection

3. To create a new database connection, in the Connections Navigator, right-click Connections.
   Select New Connection from the menu. The New/Select Database Connection dialog box appears.

4. Create a database connection using the following information:
   a. Connection Name: myconnection.
   b. Username: oraxx where xx is the number of your PC (Ask your instructor to assign you one ora account out of the ora1-ora20 range of accounts.).
   c. Password: oraxx
   d. Hostname: Enter the host name of the machine where your database server is running.
   e. Port: 1521
   f. SID: ORCL
   g. Ensure you select the Save Password check box.
Practice I (continued)

Testing and Connecting Using the Oracle SQL Developer Database Connection
5. Test the new connection.
6. If the status is Success, connect to the database using this new connection.

Browsing the Tables in the Connections Navigator
7. In the Connections Navigator, view the objects available to you in the Tables node. Verify that the following tables are present:
   - COUNTRIES
   - DEPARTMENTS
   - EMPLOYEES
   - JOB_GRADES
   - JOB_HISTORY
   - JOBS
   - LOCATIONS
   - REGIONS
8. Browse the structure of the EMPLOYEES table.
9. View the data of the DEPARTMENTS table.

Opening a SQL Worksheet
10. Open a new SQL Worksheet. Examine the shortcut icons available for the SQL Worksheet.
Retrieving Data Using the SQL SELECT Statement
Objectives

After completing this lesson, you should be able to do the following:

- List the capabilities of SQL `SELECT` statements
- Execute a basic `SELECT` statement

Objectives

To extract data from the database, you need to use the SQL `SELECT` statement. However, you may need to restrict the columns that are displayed. This lesson describes all the SQL statements that are needed to perform these actions. Further, you may want to create `SELECT` statements that can be used more than once.
Lesson Agenda

- Basic `SELECT` statement
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command
Capabilities of SQL SELECT Statements

A SELECT statement retrieves information from the database. With a SELECT statement, you can use the following capabilities:

- **Projection**: Select the columns in a table that are returned by a query. Select as few or as many of the columns as required.
- **Selection**: Select the rows in a table that are returned by a query. Various criteria can be used to restrict the rows that are retrieved.
- **Joining**: Bring together data that is stored in different tables by specifying the link between them. SQL joins are covered in more detail in the lesson titled “Displaying Data from Multiple Tables.”
Basic SELECT Statement

In its simplest form, a SELECT statement must include the following:

- A **SELECT** clause, which specifies the columns to be displayed
- A **FROM** clause, which identifies the table containing the columns that are listed in the **SELECT** clause

In the syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...} FROM table;
```

- **SELECT** identifies the columns to be displayed.
- **FROM** identifies the table containing those columns.

### Note
Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- A **keyword** refers to an individual SQL element.
  For example, **SELECT** and **FROM** are keywords.

- A **clause** is a part of a SQL statement.
  For example, **SELECT employee_id, last_name**, and so on is a clause.

- A **statement** is a combination of two or more clauses.
  For example, **SELECT * FROM employees** is a SQL statement.
Selecting All Columns

You can display all columns of data in a table by following the `SELECT` keyword with an asterisk (`*`). In the example in the slide, the department table contains four columns: `DEPARTMENT_ID`, `DEPARTMENT_NAME`, `MANAGER_ID`, and `LOCATION_ID`. The table contains eight rows, one for each department.

You can also display all columns in the table by listing all the columns after the `SELECT` keyword. For example, the following SQL statement (like the example in the slide) displays all columns and all rows of the `DEPARTMENTS` table:

```
SELECT department_id, department_name, manager_id, location_id
FROM departments;
```

Note: In SQL Developer, you can enter your SQL statement in a SQL Worksheet and click the “Execute Statement” icon or press [F9] to execute the statement. The output displayed in the Results tabbed page appears as shown in the slide.
Selecting Specific Columns

You can use the `SELECT` statement to display specific columns of the table by specifying the column names, separated by commas. The example in the slide displays all the department numbers and location numbers from the `DEPARTMENTS` table.

In the `SELECT` clause, specify the columns that you want in the order in which you want them to appear in the output. For example, to display location before department number (from left to right), you use the following statement:

```sql
SELECT location_id, department_id
FROM departments;
```

<table>
<thead>
<tr>
<th>LOCATION_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1700</td>
</tr>
<tr>
<td>2</td>
<td>1800</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>1400</td>
</tr>
</tbody>
</table>

...
Writing SQL Statements

- SQL statements are not case-sensitive.
- SQL statements can be entered on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.
- In SQL Developer, SQL statements can optionally be terminated by a semicolon (;). Semicolons are required when you execute multiple SQL statements.
- In SQL*Plus, you are required to end each SQL statement with a semicolon (;).

Executing SQL Statements

In SQL Developer, click the Run Script icon or press [F5] to run the command or commands in the SQL Worksheet. You can also click the Execute Statement icon or press [F9] to run a SQL statement in the SQL Worksheet. The Execute Statement icon executes the statement at the mouse pointer in the Enter SQL Statement box while the Run Script icon executes all the statements in the Enter SQL Statement box. The Execute Statement icon displays the output of the query on the Results tabbed page while the Run Script icon emulates the SQL*Plus display and shows the output on the Script Output tabbed page.

In SQL*Plus, terminate the SQL statement with a semicolon, and then press [Enter] to run the command.
Column Heading Defaults

• SQL Developer:
  – Default heading alignment: Left-aligned
  – Default heading display: Uppercase

• SQL*Plus:
  – Character and Date column headings are left-aligned.
  – Number column headings are right-aligned.
  – Default heading display: Uppercase

In SQL Developer, column headings are displayed in uppercase and are left-aligned.

```
SELECT last_name, hire_date, salary
FROM employees;
```

<table>
<thead>
<tr>
<th></th>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>King</td>
<td>17-JUN-87</td>
<td>24000</td>
</tr>
<tr>
<td>2</td>
<td>Kochhar</td>
<td>21-SEP-89</td>
<td>17000</td>
</tr>
<tr>
<td>3</td>
<td>De Haan</td>
<td>13-JAN-93</td>
<td>17000</td>
</tr>
<tr>
<td>4</td>
<td>Hunold</td>
<td>03-JAN-90</td>
<td>9000</td>
</tr>
<tr>
<td>5</td>
<td>Ernst</td>
<td>21-MAY-91</td>
<td>6000</td>
</tr>
<tr>
<td>6</td>
<td>Lorentz</td>
<td>07-FEB-99</td>
<td>4200</td>
</tr>
<tr>
<td>7</td>
<td>Mourtgos</td>
<td>16-NOV-99</td>
<td>5800</td>
</tr>
<tr>
<td>8</td>
<td>Rajs</td>
<td>17-OCT-95</td>
<td>3500</td>
</tr>
</tbody>
</table>

You can override the column heading display with an alias. Column aliases are covered later in this lesson.
Lesson Agenda

• Basic SELECT statement
• Arithmetic expressions and NULL values in the SELECT statement
• Column Aliases
• Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
• DESCRIBE command
Arithmetic Expressions

Create expressions with number and date data by using arithmetic operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Arithmetic Expressions

You may need to modify the way in which data is displayed, or you may want to perform calculations, or look at what-if scenarios. All these are possible using arithmetic expressions. An arithmetic expression can contain column names, constant numeric values, and the arithmetic operators.

Arithmetic Operators

The slide lists the arithmetic operators that are available in SQL. You can use arithmetic operators in any clause of a SQL statement (except the FROM clause).

Note: With the DATE and TIMESTAMP data types, you can use the addition and subtraction operators only.
Using Arithmetic Operators

The example in the slide uses the addition operator to calculate a salary increase of $300 for all employees. The slide also displays a `SALARY+300` column in the output.

Note that the resultant calculated column, `SALARY+300`, is not a new column in the `EMPLOYEES` table; it is for display only. By default, the name of a new column comes from the calculation that generated it—in this case, `salary+300`.

**Note:** The Oracle server ignores blank spaces before and after the arithmetic operator.

Operator Precedence

If an arithmetic expression contains more than one operator, multiplication and division are evaluated first. If operators in an expression are of the same priority, then evaluation is done from left to right.

You can use parentheses to force the expression that is enclosed by the parentheses to be evaluated first.

**Rules of Precedence:**

- Multiplication and division occur before addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to override the default precedence or to clarify the statement.
Operator Precedence

The first example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation by multiplying the monthly salary with 12, plus a one-time bonus of $100. Note that multiplication is performed before addition.

Note: Use parentheses to reinforce the standard order of precedence and to improve clarity. For example, the expression in the slide can be written as $(12 \times \text{salary}) + 100$ with no change in the result.

Using Parentheses

You can override the rules of precedence by using parentheses to specify the desired order in which the operators are to be executed.

The second example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation as follows: adding a monthly bonus of $100 to the monthly salary, and then multiplying that subtotal with 12. Because of the parentheses, addition takes priority over multiplication.
Defining a Null Value

• Null is a value that is unavailable, unassigned, unknown, or inapplicable.
• Null is not the same as zero or a blank space.

```
SELECT last_name, job_id, salary, commission_pct
FROM employees;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_RESS</td>
<td>24000</td>
<td>(null)</td>
</tr>
<tr>
<td>Kochhar</td>
<td>AD_VF</td>
<td>17000</td>
<td>(null)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 Zookcy</td>
<td>SA_MAN</td>
<td>10500</td>
<td>0.2</td>
</tr>
<tr>
<td>13 Abot</td>
<td>SA_REP</td>
<td>11000</td>
<td>0.3</td>
</tr>
<tr>
<td>14 Taylor</td>
<td>SA_REP</td>
<td>8800</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 Hapmo</td>
<td>AC_MGR</td>
<td>12000</td>
<td>(null)</td>
</tr>
<tr>
<td>20 Getz</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
<td>(null)</td>
</tr>
</tbody>
</table>

Defining a Null Value

If a row lacks a data value for a particular column, that value is said to be *null* or to contain a null. Null is a value that is unavailable, unassigned, unknown, or inapplicable. Null is not the same as zero or a blank space. Zero is a number and blank space is a character.

Columns of any data type can contain nulls. However, some constraints (NOT NULL and PRIMARY KEY) prevent nulls from being used in the column.

In the COMMISSION_PCT column in the EMPLOYEES table, notice that only a sales manager or sales representative can earn a commission. Other employees are not entitled to earn commissions. A null represents that fact.

**Note:** By default, SQL Developer uses the literal, (null), to identify null values. However, you can set it to something more relevant to you. To do so, select Preferences from the Tools menu. In the Preferences dialog box, expand the Database node. Click Advanced Parameters and on the right pane, for the “Display Null value As,” enter the appropriate value.
Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

```
SELECT last_name, 12*salary*commission_pct
FROM   employees;
```

If any column value in an arithmetic expression is null, the result is null. For example, if you attempt to perform division by zero, you get an error. However, if you divide a number by null, the result is a null or unknown.

In the example in the slide, employee King does not get any commission. Because the \texttt{COMMISSION\_PCT} column in the arithmetic expression is null, the result is null.

For more information, see the section on “Basic Elements of Oracle SQL” in \textit{Oracle Database SQL Language Reference 11g, Release 1 (11.1)}. 

Lesson Agenda

- Basic `SELECT` statement
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command
Defining a Column Alias

A column alias:

• Renames a column heading
• Is useful with calculations
• Immediately follows the column name (There can also be the optional AS keyword between the column name and alias.)
• Requires double quotation marks if it contains spaces or special characters, or if it is case-sensitive

Defining a Column Alias

When displaying the result of a query, SQL Developer normally uses the name of the selected column as the column heading. This heading may not be descriptive and, therefore, may be difficult to understand. You can change a column heading by using a column alias.

Specify the alias after the column in the SELECT list using blank space as a separator. By default, alias headings appear in uppercase. If the alias contains spaces or special characters (such as # or $), or if it is case-sensitive, enclose the alias in double quotation marks (" ").
Using Column Aliases

The first example displays the names and the commission percentages of all the employees. Note that the optional AS keyword has been used before the column alias name. The result of the query is the same whether the AS keyword is used or not. Also, note that the SQL statement has the column aliases, name and comm, in lowercase, whereas the result of the query displays the column headings in uppercase. As mentioned in the previous slide, column headings appear in uppercase by default.

The second example displays the last names and annual salaries of all the employees. Because Annual Salary contains a space, it has been enclosed in double quotation marks. Note that the column heading in the output is exactly the same as the column alias.
Lesson Agenda

• Basic `SELECT` Statement
• Arithmetic Expressions and NULL values in `SELECT` statement
• Column Aliases
• Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
• `DESCRIBE` command
**Concatenation Operator**

A concatenation operator:
- Links columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a resultant column that is a character expression

```sql
SELECT last_name||job_id AS "Employees"
FROM employees;
```

You can link columns to other columns, arithmetic expressions, or constant values to create a character expression by using the concatenation operator (||). Columns on either side of the operator are combined to make a single output column.

In the example, `LAST_NAME` and `JOB_ID` are concatenated, and given the alias `Employees`. Note that the last name of the employee and the job code are combined to make a single output column. The **AS** keyword before the alias name makes the **SELECT** clause easier to read.

**Null Values with the Concatenation Operator**

If you concatenate a null value with a character string, the result is a character string. `LAST_NAME || NULL` results in `LAST_NAME`.

**Note:** You can also concatenate date expressions with other expressions or columns.
Literal Character Strings

- A literal is a character, a number, or a date that is included in the `SELECT` statement.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

Literal Character Strings

A literal is a character, a number, or a date that is included in the `SELECT` list. It is not a column name or a column alias. It is printed for each row returned. Literal strings of free-format text can be included in the query result and are treated the same as a column in the `SELECT` list.

Date and character literals must be enclosed within single quotation marks ('); number literals need not be enclosed in a similar manner.
UsingLiteralCharacterStrings

Using Literal Character Strings

The example in the slide displays the last names and job codes of all employees. The column has the heading Employee Details. Note the spaces between the single quotation marks in the SELECT statement. The spaces improve the readability of the output.

In the following example, the last name and salary for each employee are concatenated with a literal, to give the returned rows more meaning:

```
SELECT last_name ||': 1 Month salary = '||salary Monthly
FROM employees;
```

```
MONTHLY
1 King: 1 Month salary = 24000
2 Kochhar: 1 Month salary = 17000
3 De Haan: 1 Month salary = 17000
4 Hunold: 1 Month salary = 9000
5 Ernst: 1 Month salary = 6000
6 Lorentz: 1 Month salary = 4200
7 Mourgos: 1 Month salary = 5800
8 Rajs: 1 Month salary = 3500
```
Alternative Quote (q) Operator

- Specify your own quotation mark delimiter.
- Select any delimiter.
- Increase readability and usability.

```
SELECT department_name || ' Department' ||
    q['s Manager Id: ']
    || manager_id
AS "Department and Manager"
FROM departments;
```
Duplicate Rows

The default display of queries is all rows, including duplicate rows.

1

```sql
SELECT department_id
FROM employees;
```

![Table 1]

2

```sql
SELECT DISTINCT department_id
FROM employees;
```

![Table 2]

---

Duplicate Rows

Unless you indicate otherwise, SQL displays the results of a query without eliminating the duplicate rows. The first example in the slide displays all the department numbers from the `EMPLOYEES` table. Note that the department numbers are repeated.

To eliminate duplicate rows in the result, include the `DISTINCT` keyword in the `SELECT` clause immediately after the `SELECT` keyword. In the second example in the slide, the `EMPLOYEES` table actually contains 20 rows, but there are only seven unique department numbers in the table.

You can specify multiple columns after the `DISTINCT` qualifier. The `DISTINCT` qualifier affects all the selected columns, and the result is every distinct combination of the columns.

```sql
SELECT DISTINCT department_id, job_id
FROM employees;
```

![Table 3]
Lesson Agenda

• Basic `SELECT` statement
• Arithmetic expressions and `NULL` values in the `SELECT` statement
• Column aliases
• Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
• `DESCRIBE` command
Displaying the Table Structure

- Use the `DESCRIBE` command to display the structure of a table.
- Or, select the table in the Connections tree and use the Columns tab to view the table structure.

```
DESCRIBE tablename
```

Displaying the Table Structure

In SQL Developer, you can display the structure of a table by using the `DESCRIBE` command. The command displays the column names and the data types, and it shows you whether a column must contain data (that is, whether the column has a `NOT NULL` constraint).

In the syntax, `tablename` is the name of any existing table, view, or synonym that is accessible to the user.

Using the SQL Developer GUI interface, you can select the table in the Connections tree and use the Columns tab to view the table structure.

**Note:** The `DESCRIBE` command is supported by both SQL*Plus and SQL Developer.
Using the DESCRIBE Command

The example in the slide displays information about the structure of the `EMPLOYEES` table using the `DESCRIBE` command.

In the resulting display, `Null` indicates that the values for this column may be unknown. `NOT NULL` indicates that a column must contain data. `Type` displays the data type for a column.

The data types are described in the following table:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NUMBER (p, s)</code></td>
<td>Number value having a maximum number of digits <code>p</code>, with <code>s</code> digits to the right of the decimal point</td>
</tr>
<tr>
<td><code>VARCHAR2 (s)</code></td>
<td>Variable-length character value of maximum size <code>s</code></td>
</tr>
<tr>
<td><code>DATE</code></td>
<td>Date and time value between January 1, 4712 B.C. and December 31, A.D. 9999.</td>
</tr>
<tr>
<td><code>CHAR (s)</code></td>
<td>Fixed-length character value of size <code>s</code></td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

- Write a SELECT statement that:
  - Returns all rows and columns from a table
  - Returns specified columns from a table
  - Uses column aliases to display more descriptive column headings

```
SELECT *|{{DISTINCT} column|expression [alias],...}
FROM table;
```

SELECT Statement

In this lesson, you should have learned how to retrieve data from a database table with the SELECT statement.

```
SELECT *|{{DISTINCT} column [alias],...}
FROM table;
```

In the syntax:

- `SELECT` is a list of one or more columns
- `*` selects all columns
- `DISTINCT` suppresses duplicates
- `column|expression` selects the named column or the expression
- `alias` gives the selected columns different headings
- `FROM table` specifies the table containing the columns
Practice 1: Overview

This practice covers the following topics:
• Selecting all data from different tables
• Describing the structure of tables
• Performing arithmetic calculations and specifying column names

Practice 1: Overview

In this practice, you write simple SELECT queries. The queries cover most of the SELECT clauses and operations that you learned in this lesson.
Practice 1

Part 1

Test your knowledge:
1. The following SELECT statement executes successfully:
   ```sql
   SELECT last_name, job_id, salary AS Sal
   FROM   employees;
   ```
   True/False
2. The following SELECT statement executes successfully:
   ```sql
   SELECT *
   FROM   job_grades;
   ```
   True/False
3. There are four coding errors in the following statement. Can you identify them?
   ```sql
   SELECT employee_id, last_name
   sal x 12 ANNUAL SALARY
   FROM      employees;
   ```

Part 2

Note the following points before you begin with the practices:
- Save all your lab files at the following location: D:\labs\SQL1\labs
- Enter your SQL statements in a SQL Worksheet. To save a script in SQL Developer, make sure the required SQL worksheet is active and then from the File menu, select Save As or right-click in the SQL Worksheet and select Save file to save your SQL statement as a lab_<lessonno>_<stepno>.sql script. When you are modifying an existing script, make sure you use Save As to save it with a different filename.
- To run the query, click the Execute Statement icon in the SQL Worksheet. Alternatively, you can press [F9]. For DML and DDL statements, use the Run Script icon or press [F5].
- After you have executed the query, make sure that you do not enter your next query in the same worksheet. Open a new worksheet.

You have been hired as a SQL programmer for Acme Corporation. Your first task is to create some reports based on data from the Human Resources tables.
4. Your first task is to determine the structure of the DEPARTMENTS table and its contents.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Null</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT_ID</td>
<td>NOT NULL</td>
<td>NUMBER(4)</td>
</tr>
<tr>
<td>DEPARTMENT_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>LOCATION_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>

4 rows selected
```
5. You need to determine the structure of the EMPLOYEES table.

    | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
    |---------------|-----------------|------------|-------------|
    | 1             | 10 Administration | 200        | 1700        |
    | 2             | 20 Marketing     | 201        | 1800        |
    | 3             | 50 Shipping      | 124        | 1500        |
    | 4             | 60 IT            | 103        | 1400        |
    | 5             | 80 Sales         | 149        | 2500        |
    | 6             | 90 Executive     | 100        | 1700        |
    | 7             | 110 Accounting   | 205        | 1700        |
    | 8             | 190 Contracting  | (null)     | 1700        |

The HR department wants a query to display the last name, job code, hire date, and employee number for each employee, with the employee number appearing first. Provide an alias STARTDATE for the HIRE_DATE column. Save your SQL statement to a file named lab_01_05.sql so that you can dispatch this file to the HR department.
Practice 1 (continued)

6. Test your query in the lab_01_05.sql file to ensure that it runs correctly.

Note: After you have executed the query, make sure that you do not enter your next query in the same worksheet. Open a new worksheet.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>STARTDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 King</td>
<td>AD_PRES</td>
<td>17-JUN-87</td>
</tr>
<tr>
<td>2</td>
<td>101 Kochhar</td>
<td>AD_VP</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>3</td>
<td>102 De Haan</td>
<td>AD_VP</td>
<td>13-JAN-93</td>
</tr>
<tr>
<td>4</td>
<td>103 Hunold</td>
<td>IT_PROG</td>
<td>03-JAN-90</td>
</tr>
<tr>
<td>5</td>
<td>104 Ernst</td>
<td>IT_PROG</td>
<td>21-MAY-91</td>
</tr>
<tr>
<td>6</td>
<td>107 Lorentz</td>
<td>IT_PROG</td>
<td>07-FEB-99</td>
</tr>
<tr>
<td>7</td>
<td>124 Mourgos</td>
<td>ST_MAN</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>8</td>
<td>141 Rajs</td>
<td>ST_CLERK</td>
<td>17-OCT-95</td>
</tr>
<tr>
<td>9</td>
<td>142 Davies</td>
<td>ST_CLERK</td>
<td>29-JAN-97</td>
</tr>
<tr>
<td>10</td>
<td>143 Matos</td>
<td>ST_CLERK</td>
<td>15-MAR-98</td>
</tr>
</tbody>
</table>

... 

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>205 Higgins</td>
</tr>
<tr>
<td>20</td>
<td>206 Gietz</td>
</tr>
</tbody>
</table>

7. The HR department wants a query to display all unique job codes from the EMPLOYEES table.
Practice 1 (continued)

Part 3

If you have time, complete the following exercises:

8. The HR department wants more descriptive column headings for its report on employees. Copy the statement from lab_01_05.sql to a new SQL Worksheet. Name the column headings Emp #, Employee, Job, and Hire Date, respectively. Then run your query again.

<table>
<thead>
<tr>
<th></th>
<th>Emp #</th>
<th>Employee</th>
<th>Job</th>
<th>Hire Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>17-JUN-87</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>3</td>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>13-JAN-93</td>
</tr>
<tr>
<td>4</td>
<td>103</td>
<td>Hunold</td>
<td>IT_PROG</td>
<td>03-JAN-90</td>
</tr>
<tr>
<td>5</td>
<td>104</td>
<td>Ernst</td>
<td>IT_PROG</td>
<td>21-MAY-91</td>
</tr>
<tr>
<td>6</td>
<td>107</td>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>07-FEB-99</td>
</tr>
<tr>
<td>7</td>
<td>124</td>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>8</td>
<td>141</td>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>17-OCT-95</td>
</tr>
<tr>
<td>9</td>
<td>142</td>
<td>Davies</td>
<td>ST_CLERK</td>
<td>29-JAN-97</td>
</tr>
<tr>
<td>10</td>
<td>143</td>
<td>Matos</td>
<td>ST_CLERK</td>
<td>15-MAR-98</td>
</tr>
</tbody>
</table>

9. The HR department has requested a report of all employees and their job IDs. Display the last name concatenated with the job ID (separated by a comma and space) and name the column Employee and Title.

<table>
<thead>
<tr>
<th></th>
<th>Employee</th>
<th>Title</th>
<th>Hire Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>205 Higgins</td>
<td>AC_MGR</td>
<td>07-JUN-94</td>
</tr>
<tr>
<td>20</td>
<td>206 Gietz</td>
<td>AC_ACCOUNT</td>
<td>07-JUN-94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Abbe, SA_REP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Davies, ST_CLERK</td>
</tr>
<tr>
<td>2</td>
<td>De Haan, AD_VP</td>
</tr>
<tr>
<td>3</td>
<td>Ernst, IT_PROG</td>
</tr>
<tr>
<td>4</td>
<td>Fay, MK_REP</td>
</tr>
<tr>
<td>5</td>
<td>Gietz, AC_ACCOUNT</td>
</tr>
<tr>
<td>6</td>
<td>Grant, SA_REP</td>
</tr>
<tr>
<td>7</td>
<td>Hartstein, MK_MAN</td>
</tr>
<tr>
<td>8</td>
<td>Higgins, AC_MGR</td>
</tr>
<tr>
<td>9</td>
<td>Hunold, IT_PROG</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Whalen, AD_ASST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Zlotkey, SA_MAN</td>
</tr>
</tbody>
</table>
If you want an extra challenge, complete the following exercise:

10. To familiarize yourself with the data in the EMPLOYEES table, create a query to display all the data from that table. Separate each column output by a comma. Name the column title THE_OUTPUT.
Restricting and Sorting Data
Objectives

After completing this lesson, you should be able to do the following:

• Limit the rows that are retrieved by a query
• Sort the rows that are retrieved by a query
• Use ampersand substitution to restrict and sort output at run time

Objectives

When retrieving data from the database, you may need to do the following:

• Restrict the rows of data that are displayed
• Specify the order in which the rows are displayed

This lesson explains the SQL statements that you use to perform the actions listed above.
Lesson Agenda

- Limiting rows with:
  - The WHERE clause
  - The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL conditions
  - Logical conditions using AND, OR, and NOT operators
- Rules of precedence for operators in an expression
- Sorting rows using the ORDER BY clause
- Substitution variables
- DEFINE and VERIFY commands
Limiting Rows Using a Selection

In the example in the slide, assume that you want to display all the employees in department 90. The rows with a value of 90 in the `DEPARTMENT_ID` column are the only ones that are returned. This method of restriction is the basis of the `WHERE` clause in SQL.
Limiting the Rows that Are Selected

- Restrict the rows that are returned by using the `WHERE` clause:

```sql
SELECT * | {{DISTINCT} column | expression [ alias ], ... }
FROM table
WHERE condition(s);
```

- The `WHERE` clause follows the `FROM` clause.

Limiting the Rows that Are Selected

You can restrict the rows that are returned from the query by using the `WHERE` clause. A `WHERE` clause contains a condition that must be met and it directly follows the `FROM` clause. If the condition is true, the row meeting the condition is returned.

In the syntax:

- `WHERE` restricts the query to rows that meet a condition
- `condition` is composed of column names, expressions, constants, and a comparison operator. A condition specifies a combination of one or more expressions and logical (Boolean) operators, and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

The `WHERE` clause can compare values in columns, literal, arithmetic expressions, or functions. It consists of three elements:

- Column name
- Comparison condition
- Column name, constant, or list of values
Using the **WHERE** Clause

In the example, the `SELECT` statement retrieves the employee ID, last name, job ID, and department number of all employees who are in department 90.

**Note:** You cannot use column alias in the **WHERE** clause.
Character Strings and Dates

- Character strings and date values are enclosed with single quotation marks.
- Character values are case-sensitive and date values are format-sensitive.
- The default date display format is DD-MON-RR.

```
SELECT last_name, job_id, department_id
FROM employees
WHERE last_name = 'Whalen';
```

```
SELECT last_name
FROM employees
WHERE hire_date = '17-FEB-96';
```

Character Strings and Dates

Character strings and dates in the **WHERE** clause must be enclosed with single quotation marks ('). Number constants, however, should not be enclosed with single quotation marks.

All character searches are case-sensitive. In the following example, no rows are returned because the **EMPLOYEES** table stores all the last names in mixed case:

```
SELECT last_name, job_id, department_id
FROM employees
WHERE last_name = 'WHALEN';
```

Oracle databases store dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default date display is in the **DD-MON-RR** format.

**Note:** For details about the RR format and about changing the default date format, see the lesson titled “Using Single-Row Functions to Customize Output.” Also, you learn about the use of single-row functions such as **UPPER** and **LOWER** to override the case sensitivity in the same lesson.
Comparison Operators

Comparison operators are used in conditions that compare one expression to another value or expression. They are used in the WHERE clause in the following format:

**Syntax**

```
... WHERE expr operator value
```

**Example**

```
... WHERE hire_date = '01-JAN-95'
... WHERE salary >= 6000
... WHERE last_name = 'Smith'
```

An alias cannot be used in the WHERE clause.

**Note:** The symbols ! = and ^ = can also represent the not equal to condition.

---

### Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>BETWEEN ...AND...</td>
<td>Between two values (inclusive)</td>
</tr>
<tr>
<td>IN(set)</td>
<td>Match any of a list of values</td>
</tr>
<tr>
<td>LIKE</td>
<td>Match a character pattern</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Is a null value</td>
</tr>
</tbody>
</table>
Using Comparison Operators

In the example, the SELECT statement retrieves the last name and salary from the EMPLOYEES table for any employee whose salary is less than or equal to $3,000. Note that there is an explicit value supplied to the WHERE clause. The explicit value of 3000 is compared to the salary value in the SALARY column of the EMPLOYEES table.
Range Conditions Using the BETWEEN Operator

Use the BETWEEN operator to display rows based on a range of values:

```sql
SELECT last_name, salary
FROM employees
WHERE salary BETWEEN 2500 AND 3500;
```

You can display rows based on a range of values using the BETWEEN operator. The range that you specify contains a lower limit and an upper limit.

The SELECT statement in the slide returns rows from the EMPLOYEES table for any employee whose salary is between $2,500 and $3,500.

Values that are specified with the BETWEEN operator are inclusive. However, you must specify the lower limit first.

You can also use the BETWEEN operator on character values:

```sql
SELECT last_name
FROM employees
WHERE last_name BETWEEN 'King' AND 'Smith';
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rao</td>
<td>3500</td>
</tr>
<tr>
<td>Davis</td>
<td>3100</td>
</tr>
<tr>
<td>Mato</td>
<td>2600</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
</tr>
<tr>
<td>Kochhar</td>
</tr>
<tr>
<td>Lorentz</td>
</tr>
<tr>
<td>Matos</td>
</tr>
<tr>
<td>Mourgos</td>
</tr>
<tr>
<td>Rajs</td>
</tr>
</tbody>
</table>
Membership Condition Using the IN Operator

Use the IN operator to test for values in a list:

```sql
SELECT employee_id, last_name, salary, manager_id
FROM employees
WHERE manager_id IN (100, 101, 201);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Kochhar</td>
<td>7000</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>De Hean</td>
<td>7000</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Mourgue</td>
<td>5000</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Zbikos</td>
<td>10000</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Hartstein</td>
<td>1300</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Wahlen</td>
<td>4400</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>Higgins</td>
<td>12000</td>
<td>101</td>
</tr>
<tr>
<td>8</td>
<td>Fay</td>
<td>8000</td>
<td>201</td>
</tr>
</tbody>
</table>

Membership Condition Using the IN Operator

To test for values in a specified set of values, use the IN operator. The condition defined using the IN operator is also known as the membership condition.

The slide example displays employee numbers, last names, salaries, and managers’ employee numbers for all the employees whose manager’s employee number is 100, 101, or 201.

The IN operator can be used with any data type. The following example returns a row from the EMPLOYEES table, for any employee whose last name is included in the list of names in the WHERE clause:

```sql
SELECT employee_id, manager_id, department_id
FROM employees
WHERE last_name IN ('Hartstein', 'Vargas');
```

If characters or dates are used in the list, they must be enclosed with single quotation marks (' ').

**Note:** The IN operator is internally evaluated by the Oracle server as a set of OR conditions, such as `a=value1 or a=value2 or a=value3`. Therefore, using the IN operator has no performance benefits and is used only for logical simplicity.
Pattern Matching Using the **LIKE** Operator

- Use the **LIKE** operator to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers:
  - `%` denotes zero or many characters.
  - `_` denotes one character.

```
SELECT first_name
FROM employees
WHERE first_name LIKE 'S%';
```

**Pattern Matching Using the **LIKE** Operator**

You may not always know the exact value to search for. You can select rows that match a character pattern by using the **LIKE** operator. The character pattern–matching operation is referred to as a *wildcard* search. Two symbols can be used to construct the search string.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%</code></td>
<td>Represents any sequence of zero or more characters</td>
</tr>
<tr>
<td><code>_</code></td>
<td>Represents any single character</td>
</tr>
</tbody>
</table>

The **SELECT** statement in the slide returns the first name from the **EMPLOYEES** table for any employee whose first name begins with the letter “S.” Note the uppercase “S.” Consequently, names beginning with a lowercase “s” are not returned.

The **LIKE** operator can be used as a shortcut for some **BETWEEN** comparisons. The following example displays the last names and hire dates of all employees who joined between January, 1995 and December, 1995:

```
SELECT last_name, hire_date
FROM employees
WHERE hire_date LIKE '%95';
```
Combining Wildcard Characters

- You can combine the two wildcard characters (%, _) with literal characters for pattern matching:

```sql
SELECT last_name 
FROM employees 
WHERE last_name LIKE '_o%';
```

- You can use the **ESCAPE** identifier to search for the actual % and _ symbols.

```sql
SELECT last_name 
FROM employees 
WHERE last_name LIKE 'Ho%';
```

**ESCAPE Identifier**

When you need to have an exact match for the actual % and _ characters, use the **ESCAPE** identifier. This option specifies what the escape character is. If you want to search for strings that contain SA_, you can use the following SQL statement:

```sql
SELECT employee_id, last_name, job_id 
FROM employees WHERE job_id LIKE '%SA_%' ESCAPE '\';
```

The **ESCAPE** identifier identifies the backslash (\) as the escape character. In the SQL statement, the escape character precedes the underscore (_). This causes the Oracle server to interpret the underscore literally.
Using the **NULL** Conditions

Test for nulls with the **IS NULL** operator.

```sql
SELECT last_name, manager_id
FROM employees
WHERE manager_id IS NULL;
```

Using the **NULL** Conditions

The **NULL** conditions include the **IS NULL** condition and the **IS NOT NULL** condition.

The **IS NULL** condition tests for nulls. A null value means that the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with =, because a null cannot be equal or unequal to any value. The slide example retrieves the last names and managers of all employees who do not have a manager.

Here is another example: To display the last name, job ID, and commission for all employees who are not entitled to receive a commission, use the following SQL statement:

```sql
SELECT last_name, job_id, commission_pct
FROM employees
WHERE commission_pct IS NULL;
```
## Defining Conditions Using the Logical Operators

A logical condition combines the result of two component conditions to produce a single result based on those conditions or it inverts the result of a single condition. A row is returned only if the overall result of the condition is true.

Three logical operators are available in SQL:
- **AND**
- **OR**
- **NOT**

All the examples so far have specified only one condition in the `WHERE` clause. You can use several conditions in a single `WHERE` clause using the AND and OR operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Returns TRUE if both component conditions are true</td>
</tr>
<tr>
<td>OR</td>
<td>Returns TRUE if either component condition is true</td>
</tr>
<tr>
<td>NOT</td>
<td>Returns TRUE if the condition is false</td>
</tr>
</tbody>
</table>

---

Copyright © 2007, Oracle. All rights reserved.
Using the **AND** Operator

**AND** requires both the component conditions to be true:

```sql
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary >= 10000
AND job_id LIKE '%MAN%';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>149 Zickly</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>2</td>
<td>201 Hartson</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
</tbody>
</table>

Using the **AND** Operator

In the example, both the component conditions must be true for any record to be selected. Therefore, only those employees who have a job title that contains the string ‘MAN’ and earn $10,000 or more are selected.

All character searches are case-sensitive, that is no rows are returned if ‘MAN’ is not uppercase. Further, character strings must be enclosed with quotation marks.

**AND Truth Table**

The following table shows the results of combining two expressions with **AND**:

<table>
<thead>
<tr>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Using the OR Operator

OR requires either component condition to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary >= 10000
OR job_id LIKE 'MAN%';
```

Using the OR Operator

In the example, either component condition can be true for any record to be selected. Therefore, any employee who has a job ID that contains the string ‘MAN’ or earns $10,000 or more is selected.

OR Truth Table

The following table shows the results of combining two expressions with OR:

<table>
<thead>
<tr>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Using the **NOT** Operator

The slide example displays the last name and job ID of all employees whose job ID is *not* IT_PROG, ST_CLERK, or SA_REP.

**NOT Truth Table**

The following table shows the result of applying the **NOT** operator to a condition:

<table>
<thead>
<tr>
<th>NOT</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The **NOT** operator can also be used with other SQL operators, such as BETWEEN, LIKE, and NULL.

```sql
... WHERE job_id    NOT  IN ('AC_ACCOUNT', 'AD_VP')
... WHERE salary    NOT  BETWEEN  10000 AND  15000
... WHERE last_name NOT  LIKE '%A%'
... WHERE commission_pct  IS   NOT  NULL
```
Lesson Agenda

• Limiting rows with:
  – The WHERE clause
  – The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  – Logical conditions using AND, OR, and NOT operators

• Rules of precedence for operators in an expression
• Sorting rows using the ORDER BY clause
• Substitution variables
• DEFINE and VERIFY commands
Rules of Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Arithmetic operators</td>
</tr>
<tr>
<td>2</td>
<td>Concatenation operator</td>
</tr>
<tr>
<td>3</td>
<td>Comparison conditions</td>
</tr>
<tr>
<td>4</td>
<td>IS [NOT] NULL, LIKE, [NOT] IN</td>
</tr>
<tr>
<td>5</td>
<td>[NOT] BETWEEN</td>
</tr>
<tr>
<td>6</td>
<td>Not equal to</td>
</tr>
<tr>
<td>7</td>
<td>NOT logical condition</td>
</tr>
<tr>
<td>8</td>
<td>AND logical condition</td>
</tr>
<tr>
<td>9</td>
<td>OR logical condition</td>
</tr>
</tbody>
</table>

You can use parentheses to override rules of precedence.

Rules of Precedence

The rules of precedence determine the order in which expressions are evaluated and calculated. The table in the slide lists the default order of precedence. However, you can override the default order by using parentheses around the expressions that you want to calculate first.
Rules of Precedence

Rules of Precedence (continued)

1. Precedence of the AND Operator: Example

In this example, there are two conditions:
   • The first condition is that the job ID is AD_PRES and the salary is greater than $15,000.
   • The second condition is that the job ID is SA_REP.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a president and earns more than $15,000, or if the employee is a sales representative.”

2. Using Parentheses: Example

In this example, there are two conditions:
   • The first condition is that the job ID is AD_PRES or SA_REP.
   • The second condition is that the salary is greater than $15,000.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a president or a sales representative, and if the employee earns more than $15,000.”
Lesson Agenda

• Limiting rows with:
  – The WHERE clause
  – The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  – Logical conditions using AND, OR, and NOT operators
• Rules of precedence for operators in an expression
• Sorting rows using the ORDER BY clause
• Substitution variables
• DEFINE and VERIFY commands
Using the ORDER BY Clause

• Sort retrieved rows with the ORDER BY clause:
  – ASC: Ascending order, default
  – DESC: Descending order

• The ORDER BY clause comes last in the SELECT statement:

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date;
```

Using the ORDER BY Clause

The order of rows that are returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. However, if you use the ORDER BY clause, it must be the last clause of the SQL statement. Further, you can specify an expression, an alias, or a column position as the sort condition.

**Syntax**

```
SELECT expr
FROM table
[WHERE condition(s)]
[ORDER BY {column, expr, numeric_position} [ASC|DESC]];
```

In the syntax:

- ORDER BY specifies the order in which the retrieved rows are displayed
- ASC orders the rows in ascending order (this is the default order)
- DESC orders the rows in descending order

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle server may not fetch rows in the same order for the same query twice. Use the ORDER BY clause to display the rows in a specific order.

**Note:** Use the keywords NULLS FIRST or NULLS LAST to specify whether returned rows containing null values should appear first or last in the ordering sequence.
Sorting

- Sorting in descending order:

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date DESC ;
```

- Sorting by column alias:

```
SELECT employee_id, last_name, salary*12 as annsal
FROM employees
ORDER BY annsal ;
```

Sorting

The default sort order is ascending:
- Numeric values are displayed with the lowest values first (for example, 1 to 999).
- Date values are displayed with the earliest value first (for example, 01-JAN-92 before 01-JAN-95).
- Character values are displayed in the alphabetical order (for example, “A” first and “Z” last).
- Null values are displayed last for ascending sequences and first for descending sequences.
- You can also sort by a column that is not in the SELECT list.

Examples:
1. To reverse the order in which the rows are displayed, specify the DESC keyword after the column name in the ORDER BY clause. The slide example sorts the result by the most recently hired employee.
2. You can also use a column alias in the ORDER BY clause. The slide example sorts the data by annual salary.
Sorting

- Sorting by using the column’s numeric position:

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY 3;
```

- Sorting by multiple columns:

```
SELECT last_name, department_id, salary
FROM employees
ORDER BY department_id, salary DESC;
```

Sorting (continued)

Examples:

3. You can sort query results by specifying the numeric position of the column in the `SELECT` clause. The slide example sorts the result by the `department_id` as this column is at the third position in the `SELECT` clause.

4. You can sort query results by more than one column. The sort limit is the number of columns in the given table. In the `ORDER BY` clause, specify the columns and separate the column names using commas. If you want to reverse the order of a column, specify `DESC` after its name.
Lesson Agenda

• Limiting rows with:
  – The WHERE clause
  – The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  – Logical conditions using AND, OR, and NOT operators
• Rules of precedence for operators in an expression
• Sorting rows using the ORDER BY clause
• Substitution variables
• DEFINE and VERIFY commands
Substitution Variables

So far, all the SQL statements were executed with predetermined columns, conditions and their values. Suppose that you want a query that lists the employees with various jobs and not just those whose job_ID is SA_REP. You can edit the WHERE clause to provide a different value each time you run the command, but there is also an easier way.

By using a substitution variable in place of the exact values in the WHERE clause, you can run the same query for different values.

You can create reports that prompt users to supply their own values to restrict the range of data returned, by using substitution variables. You can embed substitution variables in a command file or in a single SQL statement. A variable can be thought of as a container in which values are temporarily stored. When the statement is run, the stored value is substituted.
Substitution Variables

• Use substitution variables to:
  – Temporarily store values with single-ampersand (&) and double-ampersand (&&) substitution

• Use substitution variables to supplement the following:
  – WHERE conditions
  – ORDER BY clauses
  – Column expressions
  – Table names
  – Entire SELECT statements

Substitution Variables (continued)
You can use single-ampersand (&) substitution variables to temporarily store values.
You can also predefine variables by using the DEFINE command. DEFINE creates and assigns a value to a variable.

Restricted Ranges of Data: Examples
• Reporting figures only for the current quarter or specified date range
• Reporting on data relevant only to the user requesting the report
• Displaying personnel only within a given department

Other Interactive Effects
Interactive effects are not restricted to direct user interaction with the WHERE clause. The same principles can also be used to achieve other goals, such as:
• Obtaining input values from a file rather than from a person
• Passing values from one SQL statement to another

Note: Both SQL Developer and SQL* Plus support the substitution variables and the DEFINE/UNDEFINE commands. Though SQL Developer or SQL* Plus does not support validation checks (except for data type) on user input.
Using the Single-Ampersand Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value:

```
SELECT employee_id, last_name, salary, department_id
FROM employees
WHERE employee_id = &employee_num;
```

Using the Single-Ampersand Substitution Variable

When running a report, users often want to restrict the data that is returned dynamically. SQL*Plus or SQL Developer provides this flexibility with user variables. Use an ampersand (&) to identify each variable in your SQL statement. However, you do not need to define the value of each variable.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;user_variable</td>
<td>Indicates a variable in a SQL statement; if the variable does not exist, SQL*Plus or SQL Developer prompts the user for a value (the new variable is discarded after it is used.)</td>
</tr>
</tbody>
</table>

The example in the slide creates a SQL Developer substitution variable for an employee number. When the statement is executed, SQL Developer prompts the user for an employee number and then displays the employee number, last name, salary, and department number for that employee. With the single ampersand, the user is prompted every time the command is executed if the variable does not exist.
Using the Single-Ampersand Substitution Variable (continued)

When SQL Developer detects that the SQL statement contains an ampersand, you are prompted to enter a value for the substitution variable that is named in the SQL statement. After you enter a value and click the OK button, the results are displayed in the Results tab of your SQL Developer session.
Character and Date Values with Substitution Variables

Use single quotation marks for date and character values:

```sql
SELECT last_name, department_id, salary*12
FROM   employees
WHERE  job_id = '&job_title' ;
```

In a WHERE clause, date and character values must be enclosed with single quotation marks. The same rule applies to the substitution variables. Enclose the variable with single quotation marks within the SQL statement itself.

The slide shows a query to retrieve the employee names, department numbers, and annual salaries of all employees based on the job title value of the SQL Developer substitution variable.
Specifying Column Names, Expressions, and Text

You can use the substitution variables not only in the `WHERE` clause of a SQL statement, but also as substitution for column names, expressions, or text.

**Example:**

The slide example displays the employee number, last name, job title, and any other column that is specified by the user at run time, from the `EMPLOYEES` table. For each substitution variable in the `SELECT` statement, you are prompted to enter a value, and then click OK to proceed.

If you do not enter a value for the substitution variable, you get an error when you execute the preceding statement.

**Note:** A substitution variable can be used anywhere in the `SELECT` statement, except as the first word entered at the command prompt.
Using the Double-Ampersand Substitution Variable

Use double ampersand (&&) if you want to reuse the variable value without prompting the user each time:

```
SELECT employee_id, last_name, job_id, &&column_name
FROM     employees
ORDER BY &column_name;
```

You can use the double-ampersand (&&) substitution variable if you want to reuse the variable value without prompting the user each time. The user sees the prompt for the value only once. In the example in the slide, the user is asked to give the value for the variable, column_name, only once. The value that is supplied by the user (department_id) is used for both display and ordering of data. If you run the query again, you will not be prompted for the value of the variable.

SQL Developer stores the value that is supplied by using the DEFINE command; it uses it again whenever you reference the variable name. After a user variable is in place, you need to use the UNDEFINE command to delete it:

```
UNDEFINE column_name
```
Lesson Agenda

- Limiting rows with:
  - The **WHERE** clause
  - The comparison conditions using **=**, **<=**, **BETWEEN**, **IN**, **LIKE**, and **NULL** operators
  - Logical conditions using **AND**, **OR**, and **NOT** operators
- Rules of precedence for operators in an expression
- Sorting rows using the **ORDER BY** clause
- Substitution variables
- **DEFINE** and **VERIFY** commands
Using the DEFINE Command

- Use the DEFINE command to create and assign a value to a variable.
- Use the UNDEFINE command to remove a variable.

```
DEFINE employee_num = 200

SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;

UNDEFINE employee_num
```

Using the DEFINE Command

The example shown creates a substitution variable for an employee number by using the DEFINE command. At run time, this displays the employee number, name, salary, and department number for that employee.

Because the variable is created using the SQL Developer DEFINE command, the user is not prompted to enter a value for the employee number. Instead, the defined variable value is automatically substituted in the SELECT statement.

The EMPLOYEE_NUM substitution variable is present in the session until the user undefines it or exits the SQL Developer session.
Using the **VERIFY** Command

Use the **VERIFY** command to toggle the display of the substitution variable, both before and after SQL Developer replaces substitution variables with values:

```sql
SET VERIFY ON
SELECT employee_id, last_name, salary
FROM   employees
WHERE  employee_id = &employee_num;
```

To confirm the changes in the SQL statement, use the **VERIFY** command. Setting `SET VERIFY ON` forces SQL Developer to display the text of a command after it replaces substitution variables with values. To see the **VERIFY** output, you should use the Run Script (F5) icon in the SQL Worksheet. SQL Developer displays the text of a command after it replaces substitution variables with values, in the Script Output tab as shown in the slide.

The example in the slide displays the new value of the `EMPLOYEE_ID` column in the SQL statement followed by the output.

**SQL*Plus System Variables**

SQL*Plus uses various system variables that control the working environment. One of the variables is **VERIFY**. To obtain a complete list of all the system variables, you can issue the `SHOW ALL` command on the SQL*Plus command prompt.
Summary

In this lesson, you should have learned how to:

- Use the `WHERE` clause to restrict rows of output:
  - Use the comparison conditions
  - Use the `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Apply the logical `AND`, `OR`, and `NOT` operators
- Use the `ORDER BY` clause to sort rows of output:

  ```sql
  SELECT * || {{DISTINCT} column|expression [alias],...} 
  FROM table 
  [WHERE condition(s)] 
  [ORDER BY {column, expr, alias} [ASC|DESC]] 
  ;
  ```

- Use ampersand substitution to restrict and sort output at run time

Summary

In this lesson, you should have learned about restricting and sorting rows that are returned by the `SELECT` statement. You should also have learned how to implement various operators and conditions.

By using the substitution variables, you can add flexibility to your SQL statements. This enables the queries to prompt for the filter condition for the rows during run time.
Practice 2: Overview

This practice covers the following topics:

- Selecting data and changing the order of the rows that are displayed
- Restricting rows by using the \texttt{WHERE} clause
- Sorting rows by using the \texttt{ORDER BY} clause
- Using substitution variables to add flexibility to your SQL \texttt{SELECT} statements

Practice 2: Overview

In this practice, you build more reports, including statements that use the \texttt{WHERE} clause and the \texttt{ORDER BY} clause. You make the SQL statements more reusable and generic by including the ampersand substitution.
Practice 2

The HR department needs your assistance in creating some queries.
1. Because of budget issues, the HR department needs a report that displays the last name and salary of employees who earn more than $12,000. Save your SQL statement as a file named `lab_02_01.sql`. Run your query.

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
</tr>
</tbody>
</table>
```

2. Open a new SQL Worksheet. Create a report that displays the last name and department number for employee number 176. Run the query.

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taylor</td>
<td>80</td>
</tr>
</tbody>
</table>
```

3. The HR department needs to find high-salary and low-salary employees. Modify `lab_02_01.sql` to display the last name and salary for any employee whose salary is not in the range of $5,000 to $12,000. Save your SQL statement as `lab_02_03.sql`.

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
</tr>
<tr>
<td>Lorentz</td>
<td>4200</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
</tr>
<tr>
<td>Matos</td>
<td>2600</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
</tr>
<tr>
<td>Whalen</td>
<td>4400</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
</tr>
</tbody>
</table>
```
Practice 2 (continued)

4. Create a report to display the last name, job ID, and start date for the employees with the last names of Matos and Taylor. Order the query in ascending order by the start date.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>15-MAR-98</td>
</tr>
<tr>
<td>Taylor</td>
<td>SA_REP</td>
<td>24-MAR-98</td>
</tr>
</tbody>
</table>

5. Display the last name and department number of all employees in departments 20 or 50 in ascending alphabetical order by name.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Davies</td>
<td>50</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
</tr>
<tr>
<td>Matos</td>
<td>50</td>
</tr>
<tr>
<td>Mourgos</td>
<td>50</td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
</tr>
<tr>
<td>Vargas</td>
<td>50</td>
</tr>
</tbody>
</table>

6. Modify lab_02_03.sql to display the last name and salary of employees who earn between $5,000 and $12,000, and are in department 20 or 50. Label the columns Employee and Monthly Salary, respectively. Resave lab_02_03.sql as lab_02_06.sql. Run the statement in lab_02_06.sql.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Monthly Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fay</td>
<td>6000</td>
</tr>
<tr>
<td>Mourgos</td>
<td>5800</td>
</tr>
</tbody>
</table>
Practice 2 (continued)

7. The HR department needs a report that displays the last name and hire date for all employees who were hired in 1994.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgins</td>
<td>07-JUN-94</td>
</tr>
<tr>
<td>Gietz</td>
<td>07-JUN-94</td>
</tr>
</tbody>
</table>

8. Create a report to display the last name and job title of all employees who do not have a manager.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_PRES</td>
</tr>
</tbody>
</table>

9. Create a report to display the last name, salary, and commission of all employees who earn commissions. Sort data in descending order of salary and commissions. Use the column’s numeric position in the ORDER BY clause.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>11000</td>
<td>0.3</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>0.2</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
<td>0.2</td>
</tr>
<tr>
<td>Grant</td>
<td>7000</td>
<td>0.15</td>
</tr>
</tbody>
</table>

10. Members of the HR department want to have more flexibility with the queries that you are writing. They would like a report that displays the last name and salary of employees who earn more than an amount that the user specifies after a prompt. Save this query to a file named lab_02_10.sql. If you enter 12000 when prompted, the report displays the following results:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
</tr>
</tbody>
</table>
Practice 2 (continued)

11. The HR department wants to run reports based on a manager. Create a query that prompts the user for a manager ID and generates the employee ID, last name, salary, and department for that manager’s employees. The HR department wants the ability to sort the report on a selected column. You can test the data with the following values:

manager_id = 103, sorted by last_name:

<table>
<thead>
<tr>
<th>EMPLOYEE ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>104 Ernst</td>
<td>6000</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>107 Lorentz</td>
<td>4200</td>
<td>60</td>
</tr>
</tbody>
</table>

manager_id = 201, sorted by salary:

<table>
<thead>
<tr>
<th>EMPLOYEE ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>202 Fay</td>
<td>6000</td>
<td>20</td>
</tr>
</tbody>
</table>

manager_id = 124, sorted by employee_id:

<table>
<thead>
<tr>
<th>EMPLOYEE ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>141 Rajs</td>
<td>3500</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>142 Davies</td>
<td>3100</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>143 Matos</td>
<td>2600</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>144 Vargas</td>
<td>2500</td>
<td>50</td>
</tr>
</tbody>
</table>
**Practice 2 (continued)**

If you have time, complete the following exercises:

12. Display all employee last names in which the third letter of the name is “a.”

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grant</td>
</tr>
<tr>
<td>Whalen</td>
</tr>
</tbody>
</table>

13. Display the last names of all employees who have both an “a” and an “e” in their last name.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Davies</td>
</tr>
<tr>
<td>De Haan</td>
</tr>
<tr>
<td>Hartstein</td>
</tr>
<tr>
<td>Whalen</td>
</tr>
</tbody>
</table>

If you want an extra challenge, complete the following exercises:

14. Display the last name, job, and salary for all employees whose jobs are either those of a sales representative or of a stock clerk, and whose salaries are not equal to $2,500, $3,500, or $7,000.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>SA_REP</td>
<td>11000</td>
</tr>
<tr>
<td>Taylor</td>
<td>SA_REP</td>
<td>8600</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
<td>3100</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>2600</td>
</tr>
</tbody>
</table>

15. Modify lab_02_06.sql to display the last name, salary, and commission for all employees whose commission is 20%. Resave lab_02_06.sql as lab_02_15.sql. Rerun the statement in lab_02_15.sql.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Monthly Salary</th>
<th>COMMISSION_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>0.2</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Using Single-Row Functions to Customize Output
Objectives

After completing this lesson, you should be able to do the following:

• Describe various types of functions available in SQL
• Use character, number, and date functions in SELECT statements

Objectives

Functions make the basic query block more powerful, and they are used to manipulate data values. This is the first of two lessons that explore functions. It focuses on single-row character, number, and date functions.
Lesson Agenda

- Single-row SQL functions
- Character functions
- Number functions
- Working with dates
- Date functions
SQL Functions

Functions are a very powerful feature of SQL. They can be used to do the following:

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column data types

SQL functions sometimes take arguments and always return a value.

Note: If you want to know whether a function is a SQL:2003 compliant function, refer to the Oracle Compliance To Core SQL:2003 section in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Two Types of SQL Functions

There are two types of functions:
- Single-row functions
- Multiple-row functions

Single-Row Functions
These functions operate on single rows only and return one result per row. There are different types of single-row functions. This lesson covers the following ones:
- Character
- Number
- Date
- Conversion
- General

Multiple-Row Functions
Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as group functions (covered in lesson 5 titled “Reporting Aggregated Data Using the Group Functions”).

Note: For more information and a complete list of available functions and their syntax, see the topic, Functions in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Single-Row Functions

Single-row functions:
- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression

```
function_name [(arg1, arg2,...)]
```

Single-Row Functions

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row that is returned by the query. An argument can be one of the following:
- User-supplied constant
- Variable value
- Column name
- Expression

Features of single-row functions include:
- Acting on each row that is returned in the query
- Returning one result per row
- Possibly returning a data value of a different type than the one that is referenced
- Possibly expecting one or more arguments
- Can be used in `SELECT`, `WHERE`, and `ORDER BY` clauses; can be nested

In the syntax:
```
function_name  
arg1, arg2
```

*function_name* is the name of the function
*arg1, arg2* is any argument to be used by the function. This can be represented by a column name or expression.
Single-Row Functions (continued)

This lesson covers the following single-row functions:

- **Character functions**: Accept character input and can return both character and number values
- **Number functions**: Accept numeric input and return numeric values
- **Date functions**: Operate on values of the `DATE` data type (All date functions return a value of the `DATE` data type except the `MONTHS_BETWEEN` function, which returns a number.)

The following single-row functions are discussed in the next lesson titled “Using Conversion Functions and Conditional Expressions”:

- **Conversion functions**: Convert a value from one data type to another
- **General functions**:  
  - `NVL`
  - `NVL2`
  - `NULLIF`
  - `COALESCE`
  - `CASE`
  - `DECODE`
Lesson Agenda

• Single-row SQL functions
• Character functions
• Number functions
• Working with dates
• Date functions
Character Functions

Character Functions

Single-row character functions accept character data as input and can return both character and numeric values. Character functions can be divided into the following:

- Case-conversion functions
- Character-manipulation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER(column</td>
<td>expression)</td>
</tr>
<tr>
<td>UPPER(column</td>
<td>expression)</td>
</tr>
<tr>
<td>INITCAP(column</td>
<td>expression)</td>
</tr>
<tr>
<td>CONCAT(column1</td>
<td>expression1, column2</td>
</tr>
<tr>
<td>SUBSTR(column</td>
<td>expression,m[,n])</td>
</tr>
</tbody>
</table>

Note: The functions discussed in this lesson are only some of the available functions.
Character Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH(column</td>
<td>expression)</td>
</tr>
<tr>
<td>INSTR(column</td>
<td>expression, 'string', [m], [n])</td>
</tr>
<tr>
<td>LPAD(column</td>
<td>expression, n, 'string')</td>
</tr>
<tr>
<td>RPAD(column</td>
<td>expression, n, 'string')</td>
</tr>
<tr>
<td>TRIM(leading</td>
<td>trailing</td>
</tr>
<tr>
<td>REPLACE(text, search_string, replacement_string)</td>
<td>Searches a text expression for a character string and, if found, replaces it with a specified replacement string</td>
</tr>
</tbody>
</table>

**Note:** Some of the functions that are fully or partially SQL:2003 compliant are:

- UPPER
- LOWER
- TRIM
- LENGTH
- SUBSTR
- INSTR

Refer to the *Oracle Compliance To Core SQL:2003* section in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)* for more information.
Case-Conversion Functions

These functions convert the case for character strings:

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER('SQL Course')</td>
<td>sql course</td>
</tr>
<tr>
<td>UPPER('SQL Course')</td>
<td>SQL COURSE</td>
</tr>
<tr>
<td>INITCAP('SQL Course')</td>
<td>Sql Course</td>
</tr>
</tbody>
</table>

Case-Conversion Functions

LOWER, UPPER, and INITCAP are the three case-conversion functions.
- LOWER: Converts mixed-case or uppercase character strings to lowercase
- UPPER: Converts mixed-case or lowercase character strings to uppercase
- INITCAP: Converts the first letter of each word to uppercase and the remaining letters to lowercase

```sql
SELECT 'The job id for ''||UPPER(last_name)||'' is ''||LOWER(job_id) AS "EMPLOYEE DETAILS"
FROM employees;
```

<table>
<thead>
<tr>
<th>EMPLOYEE DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 The job id for ABEL is sa_rep</td>
</tr>
<tr>
<td>2 The job id for DAVIES is st_clerk</td>
</tr>
<tr>
<td>3 The job id for DE HAAN is ad_vp</td>
</tr>
</tbody>
</table>

...  
19 The job id for WHALEN is ad_asst 
20 The job id for ZLOTKEY is sa_man
Using Case-Conversion Functions

Display the employee number, name, and department number for employee Higgins:

```sql
SELECT employee_id, last_name, department_id
FROM employees
WHERE last_name = 'higgins';
```

0 rows selected

```sql
SELECT employee_id, last_name, department_id
FROM employees
WHERE LOWER(last_name) = 'higgins';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Higgins</td>
<td>110</td>
</tr>
</tbody>
</table>

Using Case-Conversion Functions

The slide example displays the employee number, name, and department number of employee Higgins.

The `WHERE` clause of the first SQL statement specifies the employee name as `higgins`. Because all the data in the `EMPLOYEES` table is stored in proper case, the name `higgins` does not find a match in the table, and no rows are selected.

The `WHERE` clause of the second SQL statement specifies that the employee name in the `EMPLOYEES` table is compared to `higgins`, converting the `LAST_NAME` column to lowercase for comparison purposes. Because both names are now lowercase, a match is found and one row is selected. The `WHERE` clause can be rewritten in the following manner to produce the same result:

```sql
WHERE last_name = 'Higgins'
```

The name in the output appears as it was stored in the database. To display the name in uppercase, use the `UPPER` function in the `SELECT` statement.

```sql
SELECT employee_id, UPPER(last_name), department_id
FROM employees
WHERE INITCAP(last_name) = 'Higgins';
```
Character-Manipulation Functions

These functions manipulate character strings:

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT('Hello', 'World')</td>
<td>HelloWorld</td>
</tr>
<tr>
<td>SUBSTR('HelloWorld',1,5)</td>
<td>Hello</td>
</tr>
<tr>
<td>LENGTH('HelloWorld')</td>
<td>10</td>
</tr>
<tr>
<td>INSTR('HelloWorld', 'W')</td>
<td>6</td>
</tr>
<tr>
<td>LPAD(salary,10,'**')</td>
<td>******24000</td>
</tr>
<tr>
<td>RPAD(salary, 10, '**')</td>
<td>24000*****</td>
</tr>
<tr>
<td>REPLACE ('JACK and JUE', 'J', 'BL')</td>
<td>BLACK and BLUE</td>
</tr>
<tr>
<td>TRIM('H' FROM 'HelloWorld')</td>
<td>elloWorld</td>
</tr>
</tbody>
</table>

**Character-Manipulation Functions**

CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, and TRIM are the character-manipulation functions that are covered in this lesson.

- **CONCAT**: Joins values together (You are limited to using two parameters with CONCAT.)
- **SUBSTR**: Extracts a string of determined length
- **LENGTH**: Shows the length of a string as a numeric value
- **INSTR**: Finds the numeric position of a named character
- **LPAD**: Returns an expression left-padded to the length of \( n \) characters with a character expression
- **RPAD**: Returns an expression right-padded to the length of \( n \) characters with a character expression
- **TRIM**: Trims leading or trailing characters (or both) from a character string (If `trim_character` or `trim_source` is a character literal, you must enclose it within single quotation marks.)

**Note**: You can use functions such as `UPPER` and `LOWER` with ampersand substitution. For example, use `UPPER('&job_title')` so that the user does not have to enter the job title in a specific case.
Using the Character-Manipulation Functions

The slide example displays employee first names and last names joined together, the length of the employee last name, and the numeric position of the letter “a” in the employee last name for all employees who have the string, REP, contained in the job ID starting at the fourth position of the job ID.

Example:
Modify the SQL statement in the slide to display the data for those employees whose last names end with the letter “n.”

```sql
SELECT employee_id, CONCAT(first_name, last_name) NAME, 
LENGTH (last_name), INSTR(last_name, 'a') "Contains 'a'?"
FROM employees
WHERE SUBSTR(last_name, -1, 1) = 'n';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>NAME</th>
<th>LENGTH(LAST_NAME)</th>
<th>Contains 'a'?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LexDe Haan</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>JenniferVHalen</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>MichaelHartstein</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>
Lesson Agenda

• Single-row SQL functions
• Character functions
• Number functions
• Working with dates
• Date Functions
Number Functions

Number functions accept numeric input and return numeric values. This section describes some of the number functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND(column</td>
<td>expression, n)</td>
</tr>
<tr>
<td>TRUNC(column</td>
<td>expression, n)</td>
</tr>
<tr>
<td>MOD(m,n)</td>
<td>Returns the remainder of m divided by n</td>
</tr>
</tbody>
</table>

**Note:** This list contains only some of the available number functions.

For more information, see the section on *Numeric Functions* in *Oracle Database SQL Language Reference 11g, Release 1 (11.1).*
Using the `ROUND` Function

The `ROUND` function rounds the column, expression, or value to \( n \) decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places. Conversely, if the second argument is \(-2\), the value is rounded to two decimal places to the left (rounded to the nearest unit of 100).

The `ROUND` function can also be used with date functions. You will see examples later in this lesson.

**DUAL Table**

The `DUAL` table is owned by the user `SYS` and can be accessed by all users. It contains one column, `DUMMY`, and one row with the value `X`. The `DUAL` table is useful when you want to return a value only once (for example, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data). The `DUAL` table is generally used for completeness of the `SELECT` clause syntax, because both `SELECT` and `FROM` clauses are mandatory, and several calculations do not need to select from the actual tables.
Using the **TRUNC** Function

The **TRUNC** function truncates the column, expression, or value to \( n \) decimal places.

The **TRUNC** function works with arguments similar to those of the **ROUND** function. If the second argument is 0 or is missing, the value is truncated to zero decimal places. If the second argument is 2, the value is truncated to two decimal places. Conversely, if the second argument is \(-2\), the value is truncated to two decimal places to the left. If the second argument is \(-1\), the value is truncated to one decimal place to the left.

Like the **ROUND** function, the **TRUNC** function can be used with date functions.
Using the **MOD** Function

For all employees with the job title of Sales Representative, calculate the remainder of the salary after it is divided by 5,000.

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA_REP';
```

<table>
<thead>
<tr>
<th></th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>MOD(SALARY, 5000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abdal</td>
<td>11000</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Taylor</td>
<td>8600</td>
<td>3600</td>
</tr>
<tr>
<td>3</td>
<td>Grant</td>
<td>7000</td>
<td>2000</td>
</tr>
</tbody>
</table>

Using the **MOD** Function

The **MOD** function finds the remainder of the first argument divided by the second argument. The slide example calculates the remainder of the salary after dividing it by 5,000 for all employees whose job ID is **SA_REP**.

**Note:** The **MOD** function is often used to determine whether a value is odd or even.
Lesson Agenda

• Single-row SQL functions
• Character functions
• Number functions
• Working with dates
• Date functions
Working with Dates

- The Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.
- The default date display format is DD-MON-RR.
  - Enables you to store 21st-century dates in the 20th century by specifying only the last two digits of the year
  - Enables you to store 20th-century dates in the 21st century in the same way

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-88';
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>17-JUN-87</td>
</tr>
<tr>
<td>Whalen</td>
<td>17-SEP-87</td>
</tr>
</tbody>
</table>

Working with Dates

The Oracle database stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is DD-MON-RR. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

In the example in the slide, the HIRE_DATE column output is displayed in the default format DD-MON-RR. However, dates are not stored in the database in this format. All the components of the date and time are stored. So, although a HIRE_DATE such as 17-JUN-87 is displayed as day, month, and year, there is also time and century information associated with the date. The complete data might be June 17, 1987, 5:10:43 PM.
**RR Date Format**

The RR date format is similar to the YY element, but you can use it to specify different centuries. Use the RR date format element instead of YY so that the century of the return value varies according to the specified two-digit year and the last two digits of the current year. The table in the slide summarizes the behavior of the RR element.

<table>
<thead>
<tr>
<th>Current Year</th>
<th>Given Date</th>
<th>Interpreted (RR)</th>
<th>Interpreted (YY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1994</td>
<td>27-OCT-95</td>
<td>1995</td>
<td>1995</td>
</tr>
<tr>
<td>1994</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>1917</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>2017</td>
</tr>
</tbody>
</table>

### RR Date Format Table

<table>
<thead>
<tr>
<th>Current Year</th>
<th>Specified Date</th>
<th>RR Format</th>
<th>YY Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>1917</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>2017</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-95</td>
<td>1995</td>
<td>2095</td>
</tr>
</tbody>
</table>

If two digits of the current year are:

<table>
<thead>
<tr>
<th>Specified Two-Digit Year</th>
<th>0–49</th>
<th>50–99</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–49</td>
<td>The return date is in the current century</td>
<td>The return date is in the century after the current one</td>
</tr>
<tr>
<td>50–99</td>
<td>The return date is in the current century</td>
<td>The return date is in the century before the current one</td>
</tr>
</tbody>
</table>
**Oracle Date Format**

This data is stored internally as follows:

<table>
<thead>
<tr>
<th>CENTURY</th>
<th>YEAR</th>
<th>MONTH</th>
<th>DAY</th>
<th>HOUR</th>
<th>MINUTE</th>
<th>SECOND</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>87</td>
<td>06</td>
<td>17</td>
<td>17</td>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>

**Centuries and the Year 2000**

When a record with a date column is inserted into a table, the *century* information is picked up from the `SYSDATE` function. However, when the date column is displayed on the screen, the century component is not displayed (by default).

The `DATE` data type always stores year information as a four-digit number internally: two digits for the century and two digits for the year. For example, the Oracle database stores the year as 1987 or 2004, and not just as 87 or 04.
Using the **SYSDATE** Function

**SYSDATE** is a function that returns:

- Date
- Time

```sql
SELECT sysdate
FROM dual;
```

**Note:** **SYSDATE** returns the current date and time set for the operating system on which the database resides. Hence, if you are in a place in Australia and connected to a remote database in a location in the United States (US), **sysdate** function will return the US date and time. In that case, you can use the **CURRENT_DATE** function that returns the current date in the session time zone.

The **CURRENT_DATE** function and other related time zone functions are discussed in detail in the course titled *Oracle Database 11g: SQL Fundamentals II*. 
Arithmetic with Dates

- Add or subtract a number to or from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

Because the database stores dates as numbers, you can perform calculations using arithmetic operators such as addition and subtraction. You can add and subtract number constants as well as dates.

You can perform the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date + number</td>
<td>Date</td>
<td>Adds a number of days to a date</td>
</tr>
<tr>
<td>date – number</td>
<td>Date</td>
<td>Subtracts a number of days from a date</td>
</tr>
<tr>
<td>date – date</td>
<td>Number of days</td>
<td>Subtracts one date from another</td>
</tr>
<tr>
<td>date + number/24</td>
<td>Date</td>
<td>Adds a number of hours to a date</td>
</tr>
</tbody>
</table>
Using Arithmetic Operators with Dates

The example in the slide displays the last name and the number of weeks employed for all employees in department 90. It subtracts the date on which the employee was hired from the current date (SYSDATE) and divides the result by 7 to calculate the number of weeks that a worker has been employed.

Note: SYSDATE is a SQL function that returns the current date and time. Your results may differ depending on the date and time set for the operating system of your local database when you run the SQL query.

If a more current date is subtracted from an older date, the difference is a negative number.
Lesson Agenda

• Single-row SQL functions
• Character functions
• Number functions
• Working with dates
• Date functions
Date-Manipulation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTHS_BETWEEN</td>
<td>Number of months between two dates</td>
</tr>
<tr>
<td>ADD_MONTHS</td>
<td>Add calendar months to date</td>
</tr>
<tr>
<td>NEXT_DAY</td>
<td>Next day of the date specified</td>
</tr>
<tr>
<td>LAST_DAY</td>
<td>Last day of the month</td>
</tr>
<tr>
<td>ROUND</td>
<td>Round date</td>
</tr>
<tr>
<td>TRUNC</td>
<td>Truncate date</td>
</tr>
</tbody>
</table>

Date-Manipulation Functions

Date functions operate on Oracle dates. All date functions return a value of the DATE data type except MONTHS_BETWEEN, which returns a numeric value.

- **MONTHS_BETWEEN**(*date1, date2*): Finds the number of months between *date1* and *date2*. The result can be positive or negative. If *date1* is later than *date2*, the result is positive; if *date1* is earlier than *date2*, the result is negative. The noninteger part of the result represents a portion of the month.

- **ADD_MONTHS**(*date, n*): Adds *n* number of calendar months to *date*. The value of *n* must be an integer and can be negative.

- **NEXT_DAY**(*date, 'char'?): Finds the date of the next specified day of the week ('char') following *date*. The value of char may be a number representing a day or a character string.

- **LAST_DAY**(*date*): Finds the date of the last day of the month that contains *date*.

The above list is a subset of the available date functions. ROUND and TRUNC number functions can also be used to manipulate the date values as shown below:

- **ROUND**(*date[, 'fmt']?): Returns *date* rounded to the unit that is specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is rounded to the nearest day.

- **TRUNC**(*date[, 'fmt']?): Returns *date* with the time portion of the day truncated to the unit that is specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is truncated to the nearest day.

The format models are covered in detail in the next lesson titled “Using Conversion Functions and Conditional Expressions.”
Using Date Functions

In the slide example, the `ADD_MONTHS` function adds one month to the supplied date value, “31-JAN-96” and returns “29-FEB-96.” The function recognizes the year 1996 as the leap year and hence returns the last day of the February month. If you change the input date value to “31-JAN-95,” the function returns “28-FEB-95.”

For example, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and the last day of the hire month for all employees who have been employed for fewer than 100 months.

```sql
SELECT employee_id, hire_date,
       MONTHS_BETWEEN (SYSDATE, hire_date) TENURE,
       ADD_MONTHS (hire_date, 6) REVIEW,
       NEXT_DAY (hire_date, 'FRIDAY'),
       LAST_DAY(hire_date)
FROM employees
WHERE MONTHS_BETWEEN (SYSDATE, hire_date) < 100;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>HIRE_DATE</th>
<th>TENURE</th>
<th>REVIEW</th>
<th>NEXT_DAY(HIRE_DATE,FRIDAY)</th>
<th>LAST_DAY(HIRE_DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124 16-NOV-96</td>
<td>91</td>
<td>10998000</td>
<td>16-MAY-00</td>
<td>16-MAY-99</td>
</tr>
<tr>
<td>2</td>
<td>149 29-JAN-00</td>
<td>88</td>
<td>5005652</td>
<td>29-JUL-00</td>
<td>04-FEB-00</td>
</tr>
<tr>
<td>3</td>
<td>170 24-MAY-99</td>
<td>96</td>
<td>80199566</td>
<td>24-JUL-99</td>
<td>24-MAY-00</td>
</tr>
<tr>
<td>4</td>
<td>99699 07-JUN-99</td>
<td>96</td>
<td>4020228240</td>
<td>07-DEC-99</td>
<td>11-JUN-99</td>
</tr>
<tr>
<td>5</td>
<td>113 11-JUL-97</td>
<td>02</td>
<td>250294335</td>
<td>11-DEC-07</td>
<td>15-JUN-07</td>
</tr>
</tbody>
</table>
Using **ROUND** and **TRUNC** Functions with Dates

Assume \( \text{SYSDATE} = '25\text{-JUL-03}' \):

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ROUND} (\text{SYSDATE}, '\text{MONTH}') )</td>
<td>01-AUG-03</td>
</tr>
<tr>
<td>( \text{ROUND} (\text{SYSDATE}, '\text{YEAR}') )</td>
<td>01-JAN-04</td>
</tr>
<tr>
<td>( \text{TRUNC} (\text{SYSDATE}, '\text{MONTH}') )</td>
<td>01-JUL-03</td>
</tr>
<tr>
<td>( \text{TRUNC} (\text{SYSDATE}, '\text{YEAR}') )</td>
<td>01-JAN-03</td>
</tr>
</tbody>
</table>

**Using **ROUND** and **TRUNC** Functions with Dates**

The **ROUND** and **TRUNC** functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month. If the format model is month, dates 1-15 result in the first day of the current month. Dates 16-31 result in the first day of the next month. If the format model is year, months 1-6 result in January 1 of the current year. Months 7-12 result in January 1 of the next year.

**Example:**

Compare the hire dates for all employees who started in 1997. Display the employee number, hire date, and starting month using the **ROUND** and **TRUNC** functions.

```
SELECT employee_id, hire_date,
       \( \text{ROUND} (\text{hire_date}, '\text{MONTH}') \), \( \text{TRUNC} (\text{hire_date}, '\text{MONTH}') \)
FROM   employees
WHERE  hire_date LIKE '%97';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>HIRE_DATE</th>
<th>ROUND(HIRE_DATE,'MONTH')</th>
<th>TRUNC(HIRE_DATE,'MONTH')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>142 29-JAN-97</td>
<td>01-FEB-97</td>
<td>01-JAN-97</td>
</tr>
<tr>
<td>2</td>
<td>202 17-AUG-97</td>
<td>01-SEP-97</td>
<td>01-AUG-97</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

- Perform calculations on data using functions
- Modify individual data items using functions

Summary

Single-row functions can be nested to any level. Single-row functions can manipulate the following:

- Character data: LOWER, UPPER, INITCAP, CONCAT, SUBSTR, INSTR, LENGTH
- Number data: ROUND, TRUNC, MOD
- Date values: SYSDATE, MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY

Remember the following:

- Date values can also use arithmetic operators.
- ROUND and TRUNC functions can also be used with date values.

**SYSDATE and DUAL**

SYSDATE is a date function that returns the current date and time. It is customary to select SYSDATE from a dummy table called DUAL.
Practice 3: Overview

This practice covers the following topics:

- Writing a query that displays the current date
- Creating queries that require the use of numeric, character, and date functions
- Performing calculations of years and months of service for an employee

Practice 3: Overview

This practice provides a variety of exercises using different functions that are available for character, number, and date data types.
Practice 3

Part 1

1. Write a query to display the system date. Label the column as **Date**.
   
   **Note:** If your database is remotely located in a different time zone, the output will be the date for the operating system on which the database resides.

<table>
<thead>
<tr>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-MAY-07</td>
</tr>
</tbody>
</table>

2. The HR department needs a report to display the employee number, last name, salary, and salary increased by 15.5% (expressed as a whole number) for each employee. Label the column **New Salary**. Save your SQL statement in a file named **lab_03_02.sql**.

3. Run your query in the **lab_03_02.sql** file.

4. Modify your query **lab_03_02.sql** to add a column that subtracts the old salary from the new salary. Label the column **Increase**. Save the contents of the file as **lab_03_04.sql**.

   Run the revised query.
Practice 3 (continued)

5. Write a query that displays the last name (with the first letter in uppercase and all the other letters in lowercase) and the length of the last name for all employees whose name starts with the letters “J,” “A,” or “M.” Give each column an appropriate label. Sort the results by the employees’ last names.

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>4</td>
</tr>
<tr>
<td>Matos</td>
<td>5</td>
</tr>
<tr>
<td>Mourgos</td>
<td>7</td>
</tr>
</tbody>
</table>

Rewrite the query so that the user is prompted to enter a letter that the last name starts with. For example, if the user enters “H” (capitalized) when prompted for a letter, then the output should show all employees whose last name starts with the letter “H.”

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>9</td>
</tr>
<tr>
<td>Higgins</td>
<td>7</td>
</tr>
<tr>
<td>Hunold</td>
<td>6</td>
</tr>
</tbody>
</table>

Modify the query such that the case of the entered letter does not affect the output. The entered letter must be capitalized before being processed by the SELECT query.
6. The HR department wants to find the duration of employment for each employee. For each employee, display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column as `MONTHS_WORKED`. Order your results by the number of months employed. Round the number of months up to the closest whole number.

*Note:* Because this query depends on the date when it was executed, the values in the `MONTHS_WORKED` column will differ for you.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>MONTHS_WORKED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlotkey</td>
<td>88</td>
</tr>
<tr>
<td>Mourgos</td>
<td>90</td>
</tr>
<tr>
<td>Grant</td>
<td>96</td>
</tr>
<tr>
<td>Lorentz</td>
<td>100</td>
</tr>
<tr>
<td>Vargas</td>
<td>107</td>
</tr>
<tr>
<td>Taylor</td>
<td>110</td>
</tr>
<tr>
<td>Matos</td>
<td>111</td>
</tr>
<tr>
<td>Fay</td>
<td>117</td>
</tr>
<tr>
<td>Davies</td>
<td>124</td>
</tr>
<tr>
<td>Abel</td>
<td>133</td>
</tr>
<tr>
<td>Hartstein</td>
<td>135</td>
</tr>
<tr>
<td>Rajs</td>
<td>139</td>
</tr>
<tr>
<td>Higgins</td>
<td>156</td>
</tr>
<tr>
<td>Gietz</td>
<td>156</td>
</tr>
<tr>
<td>De Haan</td>
<td>173</td>
</tr>
<tr>
<td>Ernst</td>
<td>192</td>
</tr>
<tr>
<td>Hunold</td>
<td>209</td>
</tr>
<tr>
<td>Kochhar</td>
<td>212</td>
</tr>
<tr>
<td>Whalen</td>
<td>236</td>
</tr>
<tr>
<td>King</td>
<td>239</td>
</tr>
</tbody>
</table>
Practice 3 (continued)

If you have time, complete the following exercises:

7. Create a query to display the last name and salary for all employees. Format the salary to be 15 characters long, left-padded with the $ symbol. Label the column as SALARY.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>$24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>$17000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

8. Create a query that displays the first eight characters of the employees’ last names and indicates the amounts of their salaries with asterisks. Each asterisk signifies a thousand dollars. Sort the data in descending order of salary. Label the column as EMPLOYEES_AND_THEIR_SALARIES.

<table>
<thead>
<tr>
<th>EMPLOYEES_AND_THEIR_SALARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
</tr>
<tr>
<td>Kochhar</td>
</tr>
<tr>
<td>De Haan</td>
</tr>
<tr>
<td>Hartstei</td>
</tr>
<tr>
<td>Higgins</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

9. Create a query to display the last name and the number of weeks employed for all employees in department 90. Label the number of weeks column as TENURE. Truncate the number of weeks value to 0 decimal places. Show the records in descending order of the employee’s tenure. Note: The TENURE value will differ as it depends on the date on which you run the query.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>TENURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>1041</td>
</tr>
<tr>
<td>Kochhar</td>
<td>923</td>
</tr>
<tr>
<td>De Haan</td>
<td>750</td>
</tr>
</tbody>
</table>
Using Conversion Functions and Conditional Expressions
Objectives

After completing this lesson, you should be able to do the following:

• Describe various types of conversion functions that are available in SQL
• Use the `TO_CHAR`, `TO_NUMBER`, and `TO_DATE` conversion functions
• Apply conditional expressions in a `SELECT` statement

Objectives

This lesson focuses on functions that convert data from one type to another (for example, conversion from character data to numeric data) and discusses the conditional expressions in SQL `SELECT` statements.
Lesson Agenda

• Implicit and explicit data type conversion
• TO_CHAR, TO_DATE, TO_NUMBER functions
• Nesting functions
• General functions:
  – NVL
  – NVL2
  – NULLIF
  – COALESCE
• Conditional expressions:
  – CASE
  – DECODE
Conversion Functions

In addition to Oracle data types, columns of tables in an Oracle database can be defined by using the American National Standards Institute (ANSI), DB2, and SQL/DS data types. However, the Oracle server internally converts such data types to Oracle data types.

In some cases, the Oracle server receives data of one data type where it expects data of a different data type. When this happens, the Oracle server can automatically convert the data to the expected data type. This data type conversion can be done implicitly by the Oracle server or explicitly by the user.

Implicit data type conversions work according to the rules explained in the next two slides.

Explicit data type conversions are done by using the conversion functions. Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention `data type TO data type`. The first data type is the input data type and the second data type is the output.

Note: Although implicit data type conversion is available, it is recommended that you do the explicit data type conversion to ensure the reliability of your SQL statements.
Implicit Data Type Conversion

In expressions, the Oracle server can automatically convert the following:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2 or CHAR</td>
<td>NUMBER</td>
</tr>
<tr>
<td>VARCHAR2 or CHAR</td>
<td>DATE</td>
</tr>
</tbody>
</table>

Implicit Data Type Conversion

Oracle server can automatically perform data type conversion in an expression. For example, the expression `hire_date > '01-JAN-90'` results in the implicit conversion from the string `'01-JAN-90'` to a date. Therefore, a VARCHAR2 or CHAR value can be implicitly converted to a number or date data type in an expression.
Implicit Data Type Conversion

For expression evaluation, the Oracle server can automatically convert the following:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>VARCHAR2 or CHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>VARCHAR2 or CHAR</td>
</tr>
</tbody>
</table>

Implicit Data Type Conversion (continued)

In general, the Oracle server uses the rule for expressions when a data type conversion is needed. For example, the expression `grade = 2` results in the implicit conversion of the number `20000` to the string “2” because grade is a `CHAR(2)` column.

**Note:** `CHAR` to `NUMBER` conversions succeed only if the character string represents a valid number.
Explicit Data Type Conversion

SQL provides three functions to convert a value from one data type to another:

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>`TO_CHAR(number</td>
<td>date, [fmt], [nlsparsms])`</td>
</tr>
</tbody>
</table>
## Explicit Data Type Conversion (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>**TO_CHAR(number</td>
<td>date,[ fmt], [nlsparams])**</td>
</tr>
<tr>
<td><strong>TO_NUMBER(char,[fmt], [nlsparams])</strong></td>
<td>Converts a character string containing digits to a number in the format specified by the optional format model fmt. The nlsparams parameter has the same purpose in this function as in the TO_CHAR function for number conversion.</td>
</tr>
<tr>
<td><strong>TO_DATE(char,[fmt],[nlsparams])</strong></td>
<td>Converts a character string representing a date to a date value according to the fmt that is specified. If fmt is omitted, the format is DD-MON-YY. The nlsparams parameter has the same purpose in this function as in the TO_CHAR function for date conversion.</td>
</tr>
</tbody>
</table>
Explicit Data Type Conversion (continued)

Note: The list of functions mentioned in this lesson includes only some of the available conversion functions.

For more information, see the section on Conversion Functions in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Lesson Agenda

• Implicit and explicit data type conversion
• TO_CHAR, TO_DATE, TO_NUMBER functions
• Nesting functions
• General functions:
  – NVL
  – NVL2
  – NULLIF
  – COALESCE
• Conditional expressions:
  – CASE
  – DECODE
Using the `TO_CHAR` Function with Dates

`TO_CHAR(date, 'format_model')`

The format model:
- Must be enclosed with single quotation marks
- Is case-sensitive
- Can include any valid date format element
- Has an `fm` element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

Using the `TO_CHAR` Function with Dates

`TO_CHAR` converts a datetime data type to a value of `VARCHAR2` data type in the format specified by the `format_model`. A format model is a character literal that describes the format of datetime stored in a character string. For example, the datetime format model for the string '11-Nov-1999' is 'DD-Mon-YYYY'. You can use the `TO_CHAR` function to convert a date from its default format to the one that you specify.

**Guidelines**
- The format model must be enclosed with single quotation marks and is case-sensitive.
- The format model can include any valid date format element. But be sure to separate the date value from the format model with a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode `fm` element.

```sql
SELECT employee_id, TO_CHAR(hire_date, 'MM/YY') Month_Hired
FROM employees
WHERE last_name = 'Higgins';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MONTH_HIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>205</td>
<td>06/94</td>
</tr>
</tbody>
</table>
## Elements of the Date Format Model

<table>
<thead>
<tr>
<th>Element</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>Full year in numbers</td>
</tr>
<tr>
<td>YEAR</td>
<td>Year spelled out (in English)</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit value for the month</td>
</tr>
<tr>
<td>MONTH</td>
<td>Full name of the month</td>
</tr>
<tr>
<td>MON</td>
<td>Three-letter abbreviation of the month</td>
</tr>
<tr>
<td>DY</td>
<td>Three-letter abbreviation of the day of the week</td>
</tr>
<tr>
<td>DAY</td>
<td>Full name of the day of the week</td>
</tr>
<tr>
<td>DD</td>
<td>Numeric day of the month</td>
</tr>
</tbody>
</table>
Sample Format Elements of Valid Date Formats

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCC or CC</td>
<td>Century; server prefixes B.C. date with -</td>
</tr>
<tr>
<td>Years in dates YYYY or SYYYY</td>
<td>Year; server prefixes B.C. date with -</td>
</tr>
<tr>
<td>YYY or YY or Y</td>
<td>Last three, two, or one digit of the year</td>
</tr>
<tr>
<td>Y,YYY</td>
<td>Year with comma in this position</td>
</tr>
<tr>
<td>IYYYY, IYY, IY, I</td>
<td>Four-, three-, two-, or one-digit year based on the ISO standard</td>
</tr>
<tr>
<td>SYEAR or YEAR</td>
<td>Year spelled out; server prefixes B.C. date with -</td>
</tr>
<tr>
<td>BC or AD</td>
<td>Indicates B.C. or A.D. year</td>
</tr>
<tr>
<td>B.C. or A.D.</td>
<td>Indicates B.C. or A.D. year using periods</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter of year</td>
</tr>
<tr>
<td>MM</td>
<td>Month: two-digit value</td>
</tr>
<tr>
<td>MONTH</td>
<td>Name of the month padded with blanks to a length of nine characters</td>
</tr>
<tr>
<td>MON</td>
<td>Name of the month, three-letter abbreviation</td>
</tr>
<tr>
<td>RM</td>
<td>Roman numeral month</td>
</tr>
<tr>
<td>WW or W</td>
<td>Week of the year or month</td>
</tr>
<tr>
<td>DDD or DD or D</td>
<td>Day of the year, month, or week</td>
</tr>
<tr>
<td>DAY</td>
<td>Name of the day padded with blanks to a length of nine characters</td>
</tr>
<tr>
<td>DY</td>
<td>Name of the day; three-letter abbreviation</td>
</tr>
<tr>
<td>J</td>
<td>Julian day; the number of days since December 31, 4713 B.C.</td>
</tr>
<tr>
<td>IW</td>
<td>Weeks in the year from ISO standard (1 to 53)</td>
</tr>
</tbody>
</table>
Elements of the Date Format Model

- Time elements format the time portion of the date:

  | HH24 : MI : SS AM | 15 : 45 : 32 PM |

- Add character strings by enclosing them with double quotation marks:

  | DD "of" MONTH | 12 of OCTOBER |

- Number suffixes spell out numbers:

  | ddspth | fourteenth |

Use the formats that are listed in the following tables to display time information and literals, and to change numerals to spelled numbers.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM or PM</td>
<td>Meridian indicator</td>
</tr>
<tr>
<td>A.M. or P.M.</td>
<td>Meridian indicator with periods</td>
</tr>
<tr>
<td>HH or HH12 or HH24</td>
<td>Hour of day, or hour (1–12), or hour (0–23)</td>
</tr>
<tr>
<td>MI</td>
<td>Minute (0–59)</td>
</tr>
<tr>
<td>SS</td>
<td>Second (0–59)</td>
</tr>
<tr>
<td>SSSSSS</td>
<td>Seconds past midnight (0–86399)</td>
</tr>
</tbody>
</table>
### Other Formats

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ . ,</td>
<td>Punctuation is reproduced in the result.</td>
</tr>
<tr>
<td>“of the”</td>
<td>Quoted string is reproduced in the result.</td>
</tr>
</tbody>
</table>

### Specifying Suffixes to Influence Number Display

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TH</td>
<td>Ordinal number (for example, DDTH for 4TH)</td>
</tr>
<tr>
<td>SP</td>
<td>Spelled-out number (for example, DDSP for FOUR)</td>
</tr>
<tr>
<td>SPTH or THSP</td>
<td>Spelled-out ordinal numbers (for example, DDSPTH for FOURTH)</td>
</tr>
</tbody>
</table>
### Using the TO_CHAR Function with Dates

The SQL statement in the slide displays the last names and hire dates for all the employees. The hire date appears as 17 June 1987.

**Example:**

Modify the example in the slide to display the dates in a format that appears as “Seventeenth of June 1987 12:00:00 AM.”

```sql
SELECT last_name,
       TO_CHAR(hire_date, 'fmDspth "of" Month YYYY fmHH:MI:SS AM')
       HIREDATE
FROM    employees;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>17th of June 1987 12:00:00 AM</td>
</tr>
<tr>
<td>Kochhar</td>
<td>21st of September 1989 12:00:00 AM</td>
</tr>
<tr>
<td>de Hoan</td>
<td>13th of January 1993 12:00:00 AM</td>
</tr>
<tr>
<td>Hundal</td>
<td>3rd of January 1999 12:00:00 AM</td>
</tr>
<tr>
<td>Ernst</td>
<td>21st of May 1991 12:00:00 AM</td>
</tr>
<tr>
<td>Lorentz</td>
<td>7th of February 1999 12:00:00 AM</td>
</tr>
<tr>
<td>Murgas</td>
<td>15th of November 1999 12:00:00 AM</td>
</tr>
<tr>
<td>Rain</td>
<td>17th of October 1999 12:00:00 AM</td>
</tr>
<tr>
<td>Davies</td>
<td>29th of January 1997 12:00:00 AM</td>
</tr>
<tr>
<td>Matos</td>
<td>15th of March 1998 12:00:00 AM</td>
</tr>
<tr>
<td></td>
<td>19th of June 1994 12:00:00 AM</td>
</tr>
<tr>
<td></td>
<td>20th of June 1994 12:00:00 AM</td>
</tr>
</tbody>
</table>

Notice that the month follows the format model specified; in other words, the first letter is capitalized and the rest are in lowercase.
Using the `TO_CHAR` Function with Numbers

<table>
<thead>
<tr>
<th>Element</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Represents a number</td>
</tr>
<tr>
<td>0</td>
<td>Forces a zero to be displayed</td>
</tr>
<tr>
<td>$</td>
<td>Places a floating dollar sign</td>
</tr>
<tr>
<td>L</td>
<td>Uses the floating local currency symbol</td>
</tr>
<tr>
<td>.</td>
<td>Prints a decimal point</td>
</tr>
<tr>
<td>,</td>
<td>Prints a comma as a thousands indicator</td>
</tr>
</tbody>
</table>

When working with number values, such as character strings, you should convert those numbers to the character data type using the `TO_CHAR` function, which translates a value of `NUMBER` data type to `VARCHAR2` data type. This technique is especially useful with concatenation.
Using the **TO_CHAR** Function with Numbers (continued)

**Number Format Elements**

If you are converting a number to the character data type, you can use the following format elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Numeric position (number of 9s determine display width)</td>
<td>9999999</td>
<td>1234</td>
</tr>
<tr>
<td>0</td>
<td>Display leading zeros</td>
<td>09999999</td>
<td>001234</td>
</tr>
<tr>
<td>$</td>
<td>Floating dollar sign</td>
<td>$99999999</td>
<td>$1234</td>
</tr>
<tr>
<td>L</td>
<td>Floating local currency symbol</td>
<td>L99999999</td>
<td>FF1234</td>
</tr>
<tr>
<td>D</td>
<td>Returns the decimal character in the specified position. The default is a period (.).</td>
<td>99D999</td>
<td>99.99</td>
</tr>
<tr>
<td>.</td>
<td>Decimal point in position specified</td>
<td>9999999.99</td>
<td>1234.00</td>
</tr>
<tr>
<td>G</td>
<td>Returns the group separator in the specified position. You can specify multiple group separators in a number format model.</td>
<td>9,999</td>
<td>9G999</td>
</tr>
<tr>
<td>,</td>
<td>Comma in position specified</td>
<td>999,999</td>
<td>1,234</td>
</tr>
<tr>
<td>MI</td>
<td>Minus signs to right (negative values)</td>
<td>9999999MI</td>
<td>1234-</td>
</tr>
<tr>
<td>PR</td>
<td>Parenthesize negative numbers</td>
<td>9999999PR</td>
<td>&lt;1234&gt;</td>
</tr>
<tr>
<td>EEEE</td>
<td>Scientific notation (format must specify four Es)</td>
<td>99.999EEE</td>
<td>1.234E+03</td>
</tr>
<tr>
<td>U</td>
<td>Returns in the specified position the “Euro” (or other) dual currency</td>
<td>U9999</td>
<td>€1234</td>
</tr>
<tr>
<td>V</td>
<td>Multiply by 10 ( n ) times ( n = ) number of 9s after V)</td>
<td>9999V999</td>
<td>1234000</td>
</tr>
<tr>
<td>S</td>
<td>Returns the negative or positive value</td>
<td>S9999</td>
<td>-1234 or +1234</td>
</tr>
<tr>
<td>B</td>
<td>Display zero values as blank, not 0</td>
<td>B9999.99</td>
<td>1234.00</td>
</tr>
</tbody>
</table>
Using the `TO_CHAR` Function with Numbers

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY
FROM employees
WHERE last_name = 'Ernst';
```

<table>
<thead>
<tr>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6,000.00</td>
</tr>
</tbody>
</table>

Using the `TO_CHAR` Function with Numbers (continued)

- The Oracle server displays a string of number signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model.
- The Oracle server rounds the stored decimal value to the number of decimal places provided in the format model.
Using the **TO_NUMBER** and **TO_DATE** Functions

- Convert a character string to a number format using the **TO_NUMBER** function:

  ```sql
  TO_NUMBER(char[, 'format_model'])
  ```

- Convert a character string to a date format using the **TO_DATE** function:

  ```sql
  TO_DATE(char[, 'format_model'])
  ```

- These functions have an **fx** modifier. This modifier specifies the exact match for the character argument and date format model of a **TO_DATE** function.

---

Using the **TO_NUMBER** and **TO_DATE** Functions

You may want to convert a character string to either a number or a date. To accomplish this task, use the **TO_NUMBER** or **TO_DATE** functions. The format model that you select is based on the previously demonstrated format elements.

The **fx** modifier specifies the exact match for the character argument and date format model of a **TO_DATE** function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without **fx**, the Oracle server ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without **fx**, the numbers in the character argument can omit leading zeros.
Using the \texttt{TO\_NUMBER} and \texttt{TO\_DATE} Functions (continued)

Example:

Display the name and hire date for all employees who started on May 24, 1999. There are two spaces after the month \textit{May} and the number 24 in the following example. Because the \texttt{fx} modifier is used, an exact match is required and the spaces after the word \textit{May} are not recognized:

\begin{verbatim}
SELECT last_name, hire_date
FROM employees
WHERE hire_date = TO_DATE('May   24, 1999', 'fxMonth DD, YYYY');
\end{verbatim}

The error:

\begin{quote}
ORA-01858: a non-numeric character was found where a numeric was expected
01858. 00000 - "a non-numeric character was found where a numeric was expected"

*Cause:* The input data to be converted using a date format model was incorrect. The input data did not contain a number where a number was required by the format model.

*Action:* Fix the input data or the date format model to make sure the elements match in number and type. Then retry the operation.

Error at Line:1
\end{quote}
Using the TO_CHAR and TO_DATE Function with RR Date Format

To find employees hired before 1990, use the RR date format, which produces the same results whether the command is run in 1999 or now:

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM employees
WHERE hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>TO_CHAR(HIRE_DATE,'DD-MON-YYYY')</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>17-Jun-1987</td>
</tr>
<tr>
<td>Kochhar</td>
<td>21-Sep-1989</td>
</tr>
<tr>
<td>Khilsten</td>
<td>17-Sep-1987</td>
</tr>
</tbody>
</table>

Using the TO_CHAR and TO_DATE Function with RR Date Format

To find employees who were hired before 1990, the RR format can be used. Because the current year is greater than 1999, the RR format interprets the year portion of the date from 1950 to 1999. The following command, on the other hand, results in no rows being selected because the YY format interprets the year portion of the date in the current century (2090).

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-yyyy')
FROM employees
WHERE TO_DATE(hire_date, 'DD-Mon-yy') < '01-Jan-1990';
```

0 rows selected
Lesson Agenda

- Implicit and explicit data type conversion
- TO_CHAR, TO_DATE, TO_NUMBER functions
- Nesting functions
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
- Conditional expressions:
  - CASE
  - DECODE
Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the deepest level to the least deep level.

\[ F3 \left( F2 \left( F1\left( \text{col}, \text{arg1} \right), \text{arg2} \right), \text{arg3} \right) \]

Step 1 = Result 1
Step 2 = Result 2
Step 3 = Result 3

Nesting Functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. Some examples follow to show you the flexibility of these functions.
Nesting Functions (continued)

The slide example displays the last names of employees in department 60. The evaluation of the SQL statement involves three steps:

1. The inner function retrieves the first eight characters of the last name.
   \[\text{Result1} = \text{SUBSTR} (\text{LAST\_NAME}, 1, 8)\]

2. The outer function concatenates the result with \textunderscore\text{US}.
   \[\text{Result2} = \text{CONCAT} (\text{Result1}, \textunderscore\text{US})\]

3. The outermost function converts the results to uppercase.

The entire expression becomes the column heading because no column alias was given.

Example:
Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, August 13th, 1999. Order the results by hire date.

\[
\begin{align*}
\text{SELECT} & \quad \text{TO\_CHAR} (\text{NEXT\_DAY} (\text{ADD\_MONTHS} (\text{hire\_date}, 6), \text{'FRIDAY'}), \\
& \quad \text{fmDay, Month ddth, YYYY'}) \\
\text{FROM} & \quad \text{employees} \\
\text{ORDER BY} & \quad \text{hire\_date;}
\end{align*}
\]
Lesson Agenda

• Implicit and explicit data type conversion
• TO_CHAR, TO_DATE, TO_NUMBER functions
• Nesting functions
• General functions:
  – NVL
  – NVL2
  – NULLIF
  – COALESCE
• Conditional expressions:
  – CASE
  – DECODE
General Functions

The following functions work with any data type and pertain to using nulls:

- NVL (expr1, expr2)
- NVL2 (expr1, expr2, expr3)
- NULLIF (expr1, expr2)
- COALESCE (expr1, expr2, ..., exprn)

General Functions

These functions work with any data type and pertain to the use of null values in the expression list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVL</td>
<td>Converts a null value to an actual value</td>
</tr>
<tr>
<td>NVL2</td>
<td>If expr1 is not null, NVL2 returns expr2. If expr1 is null, NVL2 returns expr3. The argument expr1 can have any data type.</td>
</tr>
<tr>
<td>NULLIF</td>
<td>Compares two expressions and returns null if they are equal; returns the first expression if they are not equal</td>
</tr>
<tr>
<td>COALESCE</td>
<td>Returns the first non-null expression in the expression list</td>
</tr>
</tbody>
</table>

**Note:** For more information about the hundreds of functions available, see the section on *Functions* in *Oracle Database SQL Language Reference 11g, Release 1 (11.1).*
NVL Function

Converts a null value to an actual value:
- Data types that can be used are date, character, and number.
- Data types must match:
  - `NVL(commission_pct, 0)`
  - `NVL(hire_date, '01-JAN-97')`
  - `NVL(job_id, 'No Job Yet')`

**NVL Function**

To convert a null value to an actual value, use the **NVL** function.

**Syntax**

```
NVL(expr1, expr2)
```

In the syntax:
- `expr1` is the source value or expression that may contain a null
- `expr2` is the target value for converting the null

You can use the **NVL** function to convert any data type, but the return value is always the same as the data type of `expr1`.

**NVL Conversions for Various Data Types**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Conversion Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td><code>NVL(number_column, 9)</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>NVL(date_column, '01-JAN-95')</code></td>
</tr>
<tr>
<td>CHAR or VARCHAR2</td>
<td><code>NVL(character_column, 'Unavailable')</code></td>
</tr>
</tbody>
</table>
Using the `NVL` Function

To calculate the annual compensation of all employees, you need to multiply the monthly salary by 12 and then add the commission percentage to the result:

```
SELECT last_name, salary, (salary*12) + (salary*12*NVL(commission_pct, 0)) AS AN_SAL
FROM employees;
```

Notice that the annual compensation is calculated for only those employees who earn a commission. If any column value in an expression is null, the result is null. To calculate values for all employees, you must convert the null value to a number before applying the arithmetic operator. In the example in the slide, the `NVL` function is used to convert null values to zero.
Using the NVL2 Function

The NVL2 function examines the first expression. If the first expression is not null, then the NVL2 function returns the second expression. If the first expression is null, then the third expression is returned.

Syntax

\[
\text{NVL2}(\text{expr1}, \text{expr2}, \text{expr3})
\]

In the syntax:
- \(\text{expr1}\) is the source value or expression that may contain a null
- \(\text{expr2}\) is the value that is returned if \(\text{expr1}\) is not null
- \(\text{expr3}\) is the value that is returned if \(\text{expr1}\) is null

In the example shown in the slide, the COMMISSION_PCT column is examined. If a value is detected, the second expression of SAL+COMM is returned. If the COMMISSION_PCT column holds a null value, the third expression of SAL is returned.

The argument \(\text{expr1}\) can have any data type. The arguments \(\text{expr2}\) and \(\text{expr3}\) can have any data types except LONG. If the data types of \(\text{expr2}\) and \(\text{expr3}\) are different, the Oracle server converts \(\text{expr3}\) to the data type of \(\text{expr2}\) before comparing them, unless \(\text{expr3}\) is a null constant. In the latter case, a data type conversion is not necessary. The data type of the return value is always the same as the data type of \(\text{expr2}\), unless \(\text{expr2}\) is character data, in which case the return value’s data type is VARCHAR2.
Using the NULLIF Function

The NULLIF function compares two expressions. If they are equal, the function returns a null. If they are not equal, the function returns the first expression. However, you cannot specify the literal NULL for the first expression.

Syntax

\[
\text{NULLIF (expr1, expr2)}
\]

In the syntax:
- NULLIF compares \( \text{expr1} \) and \( \text{expr2} \). If they are equal, then the function returns null. If they are not, then the function returns \( \text{expr1} \). However, you cannot specify the literal NULL for \( \text{expr1} \).

In the example shown in the slide, the length of the first name in the EMPLOYEES table is compared to the length of the last name in the EMPLOYEES table. When the lengths of the names are equal, a null value is displayed. When the lengths of the names are not equal, the length of the first name is displayed.

Note: The NULLIF function is logically equivalent to the following CASE expression. The CASE expression is discussed on a subsequent page:

\[
\text{CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END}
\]
Using the `COALESCE` Function

- The advantage of the `COALESCE` function over the `NVL` function is that the `COALESCE` function can take multiple alternate values.
- If the first expression is not null, the `COALESCE` function returns that expression; otherwise, it does a `COALESCE` of the remaining expressions.

Using the `COALESCE` Function

The `COALESCE` function returns the first non-null expression in the list.

Syntax

\[
\text{COALESCE (expr1, expr2, ... exprn)}
\]

In the syntax:

- `expr1` returns this expression if it is not null
- `expr2` returns this expression if the first expression is null and this expression is not null
- `exprn` returns this expression if the preceding expressions are null

Note that all expressions must be of the same data type.
Using the COALESCE Function

```
SELECT last_name, employee_id,
COALESCE(TO_CHAR(commission_pct), TO_CHAR(manager_id),
'No commission and no manager')
FROM employees;
```

Using the COALESCE Function (continued)

In the example shown in the slide, if the manager_id value is not null, it is displayed. If the manager_id value is null, then the commission_pct is displayed. If the manager_id and commission_pct values are null, then “No commission and no manager” is displayed. Note, TO_CHAR function is applied so that all expressions are of the same data type.
Using the `COALESCE` Function (continued)

**Example:**

For the employees who do not get any commission, your organization wants to give a salary increment of $2,000 and for employees who get commission, the query should compute the new salary that is equal to the existing salary added to the commission amount.

```sql
SELECT last_name, salary, commission_pct,
       COALESCE((salary+(commission_pct*salary)), salary+2000, salary) "New Salary"
FROM   employees;
```

**Note:** Examine the output. For employees who do not get any commission, the New Salary column shows the salary incremented by $2,000 and for employees who get commission, the New Salary column shows the computed commission amount added to the salary.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
<th>New Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
<td>(null)</td>
<td>26000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
<td>(null)</td>
<td>19000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
<td>(null)</td>
<td>19000</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>(null)</td>
<td>11000</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
<td>(null)</td>
<td>5100</td>
</tr>
<tr>
<td>Matos</td>
<td>2600</td>
<td>(null)</td>
<td>4600</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
<td>(null)</td>
<td>4500</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>0.2</td>
<td>12600</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>0.3</td>
<td>14300</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
<td>0.2</td>
<td>10320</td>
</tr>
<tr>
<td>Grant</td>
<td>7000</td>
<td>0.15</td>
<td>8050</td>
</tr>
<tr>
<td>Whalen</td>
<td>4400</td>
<td>(null)</td>
<td>6400</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
<td>(null)</td>
<td>15000</td>
</tr>
<tr>
<td>Fay</td>
<td>6000</td>
<td>(null)</td>
<td>8000</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>(null)</td>
<td>14000</td>
</tr>
<tr>
<td>Gietz</td>
<td>8300</td>
<td>(null)</td>
<td>10300</td>
</tr>
</tbody>
</table>
Lesson Agenda

• Implicit and explicit data type conversion
• `TO_CHAR`, `TO_DATE`, `TO_NUMBER` functions
• Nesting functions
• General functions:
  – `NVL`
  – `NVL2`
  – `NULLIF`
  – `COALESCE`
• Conditional expressions:
  – `CASE`
  – `DECODE`
Conditional Expressions

- Provide the use of the **IF-THEN-ELSE** logic within a SQL statement
- Use two methods:
  - **CASE** expression
  - **DECODE** function

Conditional Expressions

The two methods that are used to implement conditional processing (**IF-THEN-ELSE** logic) in a SQL statement are the **CASE** expression and the **DECODE** function.

**Note:** The **CASE** expression complies with the ANSI SQL. The **DECODE** function is specific to Oracle syntax.
CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
CASE expr WHEN comparison_expr1 THEN return_expr1
    [WHEN comparison_expr2 THEN return_expr2
      WHEN comparison_exprn THEN return_exprn
      ELSE else_expr]
END
```

CASE Expression

CASE expressions allow you to use the IF-THEN-ELSE logic in SQL statements without having to invoke procedures.

In a simple CASE expression, the Oracle server searches for the first WHEN ... THEN pair for which expr is equal to comparison_expr and returns return_expr. If none of the WHEN ... THEN pairs meet this condition, and if an ELSE clause exists, then the Oracle server returns else_expr. Otherwise, the Oracle server returns a null. You cannot specify the literal NULL for all the return_exprs and the else_expr.

All of the expressions (expr, comparison_expr, and return_expr) must be of the same data type, which can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2.
Using the CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```sql
SELECT last_name, job_id, salary,
CASE job_id WHEN 'IT_PROG' THEN 1.10*salary
    WHEN 'ST_CLERK' THEN 1.15*salary
    WHEN 'SA_REP' THEN 1.20*salary
ELSE salary END     "REVISED_SALARY"
FROM employees;
```

Using the CASE Expression

In the SQL statement in the slide, the value of `JOB_ID` is decoded. If `JOB_ID` is `IT_PROG`, the salary increase is 10%; if `JOB_ID` is `ST_CLERK`, the salary increase is 15%; if `JOB_ID` is `SA_REP`, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written with the DECODE function.

This is an example of a searched CASE expression. In a searched CASE expression, the search occurs from left to right until an occurrence of the listed condition is found, and then it returns the return expression. If no condition is found to be true, and if an ELSE clause exists, the return expression in the ELSE clause is returned; otherwise, a NULL is returned.

```sql
SELECT last_name, salary,
(CASE WHEN salary<5000 THEN 'Low'
    WHEN salary<10000 THEN 'Medium'
    WHEN salary<20000 THEN 'Good'
ELSE 'Excellent'
END) qualified_salary
FROM employees;
```
DECODE Function

Facilitates conditional inquiries by doing the work of a CASE expression or an IF-THEN-ELSE statement:

\[
\text{DECODE}(\text{col | expression, search1, result1}
\quad [, \text{search2, result2, ...}],
\quad [, \text{default}])
\]

DECODE Function

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic that is used in various languages. The DECODE function decodes expression after comparing it to each search value. If the expression is the same as search, result is returned.

If the default value is omitted, a null value is returned where a search value does not match any of the result values.
Using the `DECODE` Function

In the SQL statement in the slide, the value of `JOB_ID` is tested. If `JOB_ID` is `IT_PROG`, the salary increase is 10%; if `JOB_ID` is `ST_CLERK`, the salary increase is 15%; if `JOB_ID` is `SA_REP`, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be expressed in pseudocode as an `IF-THEN-ELSE` statement:

```sql
IF job_id = 'IT_PROG' THEN salary = salary*1.10
IF job_id = 'ST_CLERK' THEN salary = salary*1.15
IF job_id = 'SA_REP' THEN salary = salary*1.20
ELSE salary = salary
```
Using the **DECODE** Function

Display the applicable tax rate for each employee in department 80:

```
SELECT last_name, salary,
       DECODE (TRUNC(salary/2000, 0),
              0, 0.00,
              1, 0.09,
              2, 0.20,
              3, 0.30,
              4, 0.40,
              5, 0.42,
              6, 0.44,
              0.45) TAX_RATE
FROM   employees
WHERE  department_id = 80;
```

**Using the **DECODE** Function (continued)**

This slide shows another example using the **DECODE** function. In this example, you determine the tax rate for each employee in department 80 based on the monthly salary. The tax rates are as follows:

<table>
<thead>
<tr>
<th>Monthly Salary Range</th>
<th>Tax Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.00–1,999.99</td>
<td>00%</td>
</tr>
<tr>
<td>$2,000.00–3,999.99</td>
<td>09%</td>
</tr>
<tr>
<td>$4,000.00–5,999.99</td>
<td>20%</td>
</tr>
<tr>
<td>$6,000.00–7,999.99</td>
<td>30%</td>
</tr>
<tr>
<td>$8,000.00–9,999.99</td>
<td>40%</td>
</tr>
<tr>
<td>$10,000.00–11,999.99</td>
<td>42%</td>
</tr>
<tr>
<td>$12,200.00–13,999.99</td>
<td>44%</td>
</tr>
<tr>
<td>$14,000.00 or greater</td>
<td>45%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>TAX_RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>0.42</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>0.42</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

• Alter date formats for display using functions
• Convert column data types using functions
• Use NVL functions
• Use IF-THEN-ELSE logic and other conditional expressions in a SELECT statement

Summary

Remember the following:

• Conversion functions can convert character, date, and numeric values: TO_CHAR, TO_DATE, TO_NUMBER
• There are several functions that pertain to nulls, including NVL, NVL2, NULLIF, and COALESCE.
• IF-THEN-ELSE logic can be applied within a SQL statement by using the CASE expression or the DECODE function.
Practice 4: Overview

This practice covers the following topics:
- Creating queries that use `TO_CHAR`, `TO_DATE`, and other `DATE` functions
- Creating queries that use conditional expressions such as `DECODE` and `CASE`
Practice 4

1. Create a report that produces the following for each employee:
<employee last name> earns <salary> monthly but wants <3 times salary>. Label the column Dream Salaries.

<table>
<thead>
<tr>
<th>Dream Salaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 King earns $24,000.00 monthly but wants $72,000.00.</td>
</tr>
<tr>
<td>2 Kochhar earns $17,000.00 monthly but wants $51,000.00.</td>
</tr>
<tr>
<td>3 De Haan earns $17,000.00 monthly but wants $51,000.00.</td>
</tr>
<tr>
<td>4 Hunold earns $9,000.00 monthly but wants $27,000.00.</td>
</tr>
<tr>
<td>5 Ernst earns $6,000.00 monthly but wants $18,000.00.</td>
</tr>
</tbody>
</table>

2. Display each employee’s last name, hire date, and salary review date, which is the first Monday after six months of service. Label the column REVIEW. Format the dates to appear in the format similar to “Monday, the Thirty-First of July, 2000.”

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>REVIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 King</td>
<td>17-JUN-87</td>
<td>Monday, the Twenty-First of December, 1987</td>
</tr>
<tr>
<td>2 Kochhar</td>
<td>21-SEP-89</td>
<td>Monday, the Twenty-Sixth of March, 1990</td>
</tr>
<tr>
<td>3 De Haan</td>
<td>13-JAN-93</td>
<td>Monday, the Nineteenth of July, 1993</td>
</tr>
<tr>
<td>4 Hunold</td>
<td>03-JAN-90</td>
<td>Monday, the Ninth of July, 1990</td>
</tr>
<tr>
<td>5 Ernst</td>
<td>21-MAY-91</td>
<td>Monday, the Twenty-Fifth of November, 1991</td>
</tr>
</tbody>
</table>

| 19 Higgins | 07-JUN-94 | Monday, the Twelfth of December, 1994 |
| 20 Gietz  | 07-JUN-94 | Monday, the Twelfth of December, 1994 |
Practice 4 (continued)

3. Display the last name, hire date, and day of the week on which the employee started. Label the column **DAY**. Order the results by the day of the week, starting with Monday.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grant</td>
<td>24-MAY-99</td>
<td>MONDAY</td>
</tr>
<tr>
<td>Gietz</td>
<td>07-JUN-94</td>
<td>TUESDAY</td>
</tr>
<tr>
<td>Taylor</td>
<td>24-MAR-98</td>
<td>TUESDAY</td>
</tr>
<tr>
<td>Higgins</td>
<td>07-JUN-94</td>
<td>TUESDAY</td>
</tr>
<tr>
<td>Rajs</td>
<td>17-OCT-95</td>
<td>TUESDAY</td>
</tr>
<tr>
<td>Lorentz</td>
<td>07-FEB-99</td>
<td>SUNDAY</td>
</tr>
<tr>
<td>Fay</td>
<td>17-AUG-97</td>
<td>SUNDAY</td>
</tr>
</tbody>
</table>

4. Create a query that displays the employees’ last names and commission amounts. If an employee does not earn commission, show “No Commission.” Label the column **COMM**.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>No Commission</td>
</tr>
<tr>
<td>Kochhar</td>
<td>No Commission</td>
</tr>
<tr>
<td>De Haan</td>
<td>No Commission</td>
</tr>
<tr>
<td>Hunold</td>
<td>No Commission</td>
</tr>
<tr>
<td>Ernst</td>
<td>No Commission</td>
</tr>
<tr>
<td>Lorentz</td>
<td>No Commission</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>.2</td>
</tr>
<tr>
<td>Abel</td>
<td>.3</td>
</tr>
<tr>
<td>Taylor</td>
<td>.2</td>
</tr>
<tr>
<td>Grant</td>
<td>.15</td>
</tr>
<tr>
<td>Whalen</td>
<td>No Commission</td>
</tr>
<tr>
<td>Hartstein</td>
<td>No Commission</td>
</tr>
<tr>
<td>Fay</td>
<td>No Commission</td>
</tr>
<tr>
<td>Higgins</td>
<td>No Commission</td>
</tr>
<tr>
<td>Gietz</td>
<td>No Commission</td>
</tr>
</tbody>
</table>
**Practice 4 (continued)**

If you have time, complete the following exercises:

5. Using the `DECODE` function, write a query that displays the grade of all employees based on the value of the column `JOB_ID`, using the following data:

<table>
<thead>
<tr>
<th>Job</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD_PRES</td>
<td>A</td>
</tr>
<tr>
<td>ST_MAN</td>
<td>B</td>
</tr>
<tr>
<td>IT_PROG</td>
<td>C</td>
</tr>
<tr>
<td>SA_REP</td>
<td>D</td>
</tr>
<tr>
<td>ST_CLERK</td>
<td>E</td>
</tr>
<tr>
<td>None of the above</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AC_ACCOUNT 0</td>
</tr>
<tr>
<td>2</td>
<td>AC_MGR 0</td>
</tr>
<tr>
<td>3</td>
<td>AD_ASST 0</td>
</tr>
<tr>
<td>4</td>
<td>AD_PRES A</td>
</tr>
<tr>
<td>5</td>
<td>AD_VP 0</td>
</tr>
</tbody>
</table>

...  
| 18      | ST_CLERK E |
| 19      | ST_CLERK E |
| 20      | ST_MAN B   |

6. Rewrite the statement in the preceding exercise using the `CASE` syntax.

```
<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AC_ACCOUNT 0</td>
</tr>
<tr>
<td>2</td>
<td>AC_MGR 0</td>
</tr>
<tr>
<td>3</td>
<td>AD_ASST 0</td>
</tr>
<tr>
<td>4</td>
<td>AD_PRES A</td>
</tr>
<tr>
<td>5</td>
<td>AD_VP 0</td>
</tr>
</tbody>
</table>

...  
| 18      | ST_CLERK E |
| 19      | ST_CLERK E |
| 20      | ST_MAN B   |
```
5

Reporting Aggregated Data Using the Group Functions
Objectives

After completing this lesson, you should be able to do the following:

• 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

Objectives

This lesson further addresses functions. It focuses on obtaining summary information (such as averages) for groups of rows. It discusses how to group rows in a table into smaller sets and how to specify search criteria for groups of rows.
Lesson Agenda

• Group functions:
  – Types and syntax
  – Use AVG, SUM, MIN, MAX, COUNT
  – Use DISTINCT keyword within group functions
  – NULL values in a group function

• Grouping rows:
  – GROUP BY clause
  – HAVING clause

• Nesting group functions
What Are Group Functions?

Group functions operate on sets of rows to give one result per group.

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table split into groups.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24000</td>
</tr>
<tr>
<td>2</td>
<td>17000</td>
</tr>
<tr>
<td>3</td>
<td>17000</td>
</tr>
<tr>
<td>4</td>
<td>9000</td>
</tr>
<tr>
<td>5</td>
<td>6000</td>
</tr>
<tr>
<td>6</td>
<td>4200</td>
</tr>
<tr>
<td>7</td>
<td>5800</td>
</tr>
<tr>
<td>8</td>
<td>3900</td>
</tr>
<tr>
<td>9</td>
<td>3100</td>
</tr>
<tr>
<td>10</td>
<td>2800</td>
</tr>
<tr>
<td>10</td>
<td>2800</td>
</tr>
<tr>
<td>10</td>
<td>2800</td>
</tr>
<tr>
<td>10</td>
<td>2800</td>
</tr>
</tbody>
</table>

Maximum salary in EMPLOYEES table
Types of Group Functions

Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG(([DISTINCT</td>
<td>ALL](n))</td>
</tr>
<tr>
<td>COUNT({{*</td>
<td>[DISTINCT</td>
</tr>
<tr>
<td>MAX(([DISTINCT</td>
<td>ALL](expr)</td>
</tr>
<tr>
<td>MIN(([DISTINCT</td>
<td>ALL](expr)</td>
</tr>
<tr>
<td>STDDEV(([DISTINCT</td>
<td>ALL](x)</td>
</tr>
<tr>
<td>SUM(([DISTINCT</td>
<td>ALL](n)</td>
</tr>
<tr>
<td>VARIANCE(([DISTINCT</td>
<td>ALL](x)</td>
</tr>
</tbody>
</table>
Group Functions: Syntax

The group function is placed after the `SELECT` keyword. You may have multiple group functions separated by commas.

Guidelines for using the group functions:

- `DISTINCT` makes the function consider only nonduplicate values; `ALL` makes it consider every value, including duplicates. The default is `ALL` and therefore does not need to be specified.
- The data types for the functions with an `expr` argument may be `CHAR`, `VARCHAR2`, `NUMBER`, or `DATE`.
- All group functions ignore null values. To substitute a value for null values, use the `NVL`, `NVL2`, or `COALESCE` functions.
Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
FROM   employees
WHERE  job_id LIKE '%REP%';
```

<table>
<thead>
<tr>
<th></th>
<th>AVG(salary)</th>
<th>MAX(salary)</th>
<th>MIN(salary)</th>
<th>SUM(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8150</td>
<td>11000</td>
<td>6000</td>
<td>33600</td>
</tr>
</tbody>
</table>

Using the AVG and SUM Functions

You can use the AVG, SUM, MIN, and MAX functions against the columns that can store numeric data. The example in the slide displays the average, highest, lowest, and sum of monthly salaries for all sales representatives.
Using the **MIN** and **MAX** Functions

You can use **MIN** and **MAX** for numeric, character, and date data types.

```sql
SELECT MIN(hire_date), MAX(hire_date)
FROM employees;
```

You can use the **MAX** and **MIN** functions for numeric, character, and date data types. The example in the slide displays the most junior and most senior employees.

The following example displays the employee last name that is first and the employee last name that is last in an alphabetic list of all employees:

```sql
SELECT MIN(last_name), MAX(last_name)
FROM employees;
```

<table>
<thead>
<tr>
<th>MIN(last_name)</th>
<th>MAX(last_name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>Zlotkey</td>
</tr>
</tbody>
</table>

*Note*: The **AVG**, **SUM**, **VARIANCE**, and **STDDEV** functions can be used only with numeric data types. **MAX** and **MIN** cannot be used with **LOB** or **LONG** data types.
Using the `COUNT` Function

`COUNT(*)` returns the number of rows in a table:

```sql
SELECT COUNT(*)
FROM employees
WHERE department_id = 50;
```

1 row returned.

`COUNT(expr)` returns the number of rows with non-null values for `expr`:

```sql
SELECT COUNT(commission_pct)
FROM employees
WHERE department_id = 80;
```

1 row returned.

Using the `COUNT` Function

The `COUNT` function has three formats:
- `COUNT(*)`
- `COUNT(expr)`
- `COUNT(DISTINCT expr)`

`COUNT(*)` returns the number of rows in a table that satisfy the criteria of the `SELECT` statement, including duplicate rows and rows containing null values in any of the columns. If a `WHERE` clause is included in the `SELECT` statement, `COUNT(*)` returns the number of rows that satisfy the condition in the `WHERE` clause.

In contrast, `COUNT(expr)` returns the number of non-null values that are in the column identified by `expr`.

`COUNT(DISTINCT expr)` returns the number of unique, non-null values that are in the column identified by `expr`.

Examples:
1. The example in the slide displays the number of employees in department 50.
2. The example in the slide displays the number of employees in department 80 who can earn a commission.
Using the **DISTINCT** Keyword

- `COUNT(DISTINCT expr)` returns the number of distinct non-null values of `expr`.
- To display the number of distinct department values in the **EMPLOYEES** table:

```sql
SELECT COUNT(DISTINCT department_id) FROM employees;
```

![Query Result]

Using the **DISTINCT** Keyword

Use the **DISTINCT** keyword to suppress the counting of any duplicate values in a column.

The example in the slide displays the number of distinct department values that are in the **EMPLOYEES** table.
Group Functions and Null Values

Group functions ignore null values in the column:

1. `SELECT AVG(commission_pct) FROM employees;`

```
<table>
<thead>
<tr>
<th>AVG(COMMISSION_PCT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2125</td>
</tr>
</tbody>
</table>
```

The `NVL` function forces group functions to include null values:

2. `SELECT AVG(NVL(commission_pct, 0)) FROM employees;`

```
<table>
<thead>
<tr>
<th>AVG(NVL(COMMISSION_PCT,0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0425</td>
</tr>
</tbody>
</table>
```

Group Functions and Null Values

All group functions ignore null values in the column. However, the `NVL` function forces group functions to include null values.

Examples:
1. The average is calculated based on *only* those rows in the table in which a valid value is stored in the `COMMISSION_PCT` column. The average is calculated as the total commission that is paid to all employees divided by the number of employees receiving a commission (four).
2. The average is calculated based on *all* rows in the table, regardless of whether null values are stored in the `COMMISSION_PCT` column. The average is calculated as the total commission that is paid to all employees divided by the total number of employees in the company (20).
Lesson Agenda

• Group functions:
  – Types and syntax
  – Use AVG, SUM, MIN, MAX, COUNT
  – Use DISTINCT keyword within group functions
  – NULL values in a group function

• Grouping rows:
  – GROUP BY clause
  – HAVING clause

• Nesting group functions
Creating Groups of Data

Until this point in our discussion, all group functions have treated the table as one large group of information. At times, however, you need to divide the table of information into smaller groups. This can be done by using the `GROUP BY` clause.
Creating Groups of Data: GROUP BY Clause Syntax

You can divide rows in a table into smaller groups by using the GROUP BY clause.

```sql
SELECT column, group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well, unless the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the columns in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.
Using the GROUP BY Clause

All columns in the `SELECT` list that are not in group functions must be in the `GROUP BY` clause.

```sql
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7000</td>
</tr>
<tr>
<td>2</td>
<td>9500</td>
</tr>
<tr>
<td>3</td>
<td>9500</td>
</tr>
<tr>
<td>4</td>
<td>15150</td>
</tr>
<tr>
<td>5</td>
<td>3500</td>
</tr>
<tr>
<td>6</td>
<td>8400</td>
</tr>
<tr>
<td>7</td>
<td>8400</td>
</tr>
<tr>
<td>8</td>
<td>4400</td>
</tr>
</tbody>
</table>

Using the GROUP BY Clause

When using the `GROUP BY` clause, make sure that all columns in the `SELECT` list that are not group functions are included in the `GROUP BY` clause. The example in the slide displays the department number and the average salary for each department. Here is how this `SELECT` statement, containing a `GROUP BY` clause, is evaluated:

- The `SELECT` clause specifies the columns to be retrieved, as follows:
  - Department number column in the `EMPLOYEES` table
  - The average of all salaries in the group that you specified in the `GROUP BY` clause
- The `FROM` clause specifies the tables that the database must access: the `EMPLOYEES` table
- The `WHERE` clause specifies the rows to be retrieved. Because there is no `WHERE` clause, all rows are retrieved by default.
- The `GROUP BY` clause specifies how the rows should be grouped. The rows are grouped by department number, so the `AVG` function that is applied to the salary column calculates the average salary for each department.
Using the **GROUP BY** Clause

The **GROUP BY** column does not have to be in the **SELECT** list.

```sql
SELECT AVG(salary)
FROM employees
GROUP BY department_id;
```

<table>
<thead>
<tr>
<th>Department ID</th>
<th>Average Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7000</td>
</tr>
<tr>
<td>2</td>
<td>9500</td>
</tr>
<tr>
<td>3</td>
<td>10150</td>
</tr>
<tr>
<td>4</td>
<td>8500</td>
</tr>
<tr>
<td>5</td>
<td>6400</td>
</tr>
<tr>
<td>6</td>
<td>10150</td>
</tr>
<tr>
<td>7</td>
<td>8500</td>
</tr>
<tr>
<td>8</td>
<td>6400</td>
</tr>
</tbody>
</table>

Using the **GROUP BY** Clause (continued)

The **GROUP BY** column does not have to be in the **SELECT** clause. For example, the **SELECT** statement in the slide displays the average salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful.

You can also use the group function in the **ORDER BY** clause:

```sql
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
ORDER BY AVG(salary);
```

<table>
<thead>
<tr>
<th>Department ID</th>
<th>Average Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3500</td>
</tr>
<tr>
<td>2</td>
<td>4400</td>
</tr>
<tr>
<td>3</td>
<td>6400</td>
</tr>
<tr>
<td>7</td>
<td>10150</td>
</tr>
<tr>
<td>8</td>
<td>19333.33333333</td>
</tr>
</tbody>
</table>

Oracle Database 11g: SQL Fundamentals I  5 - 16
Grouping by More than One Column

Sometimes you need to see results for groups within groups. The slide shows a report that displays the total salary that is paid to each job title in each department.

The EMPLOYEES table is grouped first by the department number and then by the job title within that grouping. For example, the four stock clerks in department 50 are grouped together, and a single result (total salary) is produced for all stock clerks in the group.
Using the **GROUP BY** Clause on Multiple Columns

You can return summary results for groups and subgroups by listing more than one **GROUP BY** column. You can determine the default sort order of the results by the order of the columns in the **GROUP BY** clause. In the example in the slide, the **SELECT** statement containing a **GROUP BY** clause is evaluated as follows:

- The **SELECT** clause specifies the column to be retrieved:
  - Department number in the **EMPLOYEES** table
  - Job ID in the **EMPLOYEES** table
  - The sum of all salaries in the group that you specified in the **GROUP BY** clause
- The **FROM** clause specifies the tables that the database must access: the **EMPLOYEES** table
- The **GROUP BY** clause specifies how you must group the rows:
  - First, the rows are grouped by the department number.
  - Second, the rows are grouped by job ID in the department number groups.

So the **SUM** function is applied to the salary column for all job IDs in each department number group.
Illegal Queries Using Group Functions

Any column or expression in the **SELECT** list that is not an aggregate function must be in the **GROUP BY** clause:

```
SELECT department_id, COUNT(last_name)
FROM   employees;
```

A **GROUP BY** clause must be added to count the last names for each `department_id`.

```
SELECT department_id, job_id, COUNT(last_name)
FROM   employees
GROUP BY department_id;
```

Either add `job_id` in the **GROUP BY** or remove the `job_id` column from the **SELECT** list.

Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (DEPARTMENT_ID) and group functions (COUNT) in the same **SELECT** statement, you must include a **GROUP BY** clause that specifies the individual items (in this case, DEPARTMENT_ID). If the **GROUP BY** clause is missing, then the error message “not a single-group group function” appears and an asterisk (*) points to the offending column. You can correct the error in the first example in the slide by adding the **GROUP BY** clause:

```
SELECT department_id, count(last_name)
FROM   employees
GROUP BY department_id;
```

Any column or expression in the **SELECT** list that is not an aggregate function must be in the **GROUP BY** clause. In the second example in the slide, job_id is neither in the **GROUP BY** clause nor is it being used by a group function, so there is a “not a **GROUP BY** expression” error. You can correct the error in the second slide example by adding `job_id` in the **GROUP BY** clause.

```
SELECT department_id, job_id, COUNT(last_name)
FROM   employees
GROUP BY department_id, job_id;
```
Illegal Queries Using Group Functions

- You cannot use the `WHERE` clause to restrict groups.
- You use the `HAVING` clause to restrict groups.
- You cannot use group functions in the `WHERE` clause.

```
SELECT department_id, AVG(salary)
FROM employees
WHERE AVG(salary) > 8000
GROUP BY department_id;
```

Illegal Queries Using Group Functions (continued)

The `WHERE` clause cannot be used to restrict groups. The `SELECT` statement in the example in the slide results in an error because it uses the `WHERE` clause to restrict the display of the average salaries of those departments that have an average salary greater than $8,000.

However, you can correct the error in the example by using the `HAVING` clause to restrict groups:

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 8000;
```
Restricting Group Results

You use the HAVING clause to restrict groups in the same way that you use the WHERE clause to restrict the rows that you select. To find the maximum salary in each of the departments that have a maximum salary greater than $10,000, you need to do the following:

1. Find the average salary for each department by grouping by department number.
2. Restrict the groups to those departments with a maximum salary greater than $10,000.
Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

```
SELECT column, group_function
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY column];
```

Restricting Group Results with the HAVING Clause

You use the HAVING clause to specify the groups that are to be displayed, thus further restricting the groups on the basis of aggregate information.

In the syntax, group_condition restricts the groups of rows returned to those groups for which the specified condition is true.

The Oracle server performs the following steps when you use the HAVING clause:

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the HAVING clause are displayed.

The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.
Using the **HAVING** Clause

The example in the slide displays the department numbers and maximum salaries for those departments with a maximum salary greater than $10,000.

You can use the **GROUP BY** clause without using a group function in the **SELECT** list. If you restrict rows based on the result of a group function, you must have a **GROUP BY** clause as well as the **HAVING** clause.

The following example displays the department numbers and average salaries for those departments with a maximum salary greater than $10,000:

```sql
SELECT department_id, AVG(salary) FROM employees GROUP BY department_id HAVING max(salary)>10000;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90 19333.3333333333333333...</td>
</tr>
<tr>
<td>2</td>
<td>20 9500</td>
</tr>
<tr>
<td>3</td>
<td>110 10150</td>
</tr>
<tr>
<td>4</td>
<td>80 10033.3333333333333333...</td>
</tr>
</tbody>
</table>

Using the **HAVING** Clause

```sql
SELECT job_id, SUM(salary) PAYROLL
FROM employees
WHERE job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING SUM(salary) > 13000
ORDER BY SUM(salary);
```

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>PAYROLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>11_FROG</td>
<td>12000</td>
</tr>
<tr>
<td>2_AD PRESS</td>
<td>24000</td>
</tr>
<tr>
<td>3_AD VP</td>
<td>36000</td>
</tr>
</tbody>
</table>

Using the **HAVING** Clause (continued)

The example in the slide displays the job ID and total monthly salary for each job that has a total payroll exceeding $13,000. The example excludes sales representatives and sorts the list by the total monthly salary.
Lesson Agenda

• Group functions:
  – Types and syntax
  – Use AVG, SUM, MIN, MAX, COUNT
  – Use DISTINCT keyword within group functions
  – NULL values in a group function

• Grouping rows:
  – GROUP BY clause
  – HAVING clause

• Nesting group functions
Nesting Group Functions

Display the maximum average salary:

```
SELECT MAX(AVG(salary))
FROM employees
GROUP BY department_id;
```

Nesting Group Functions

Group functions can be nested to a depth of two functions. The example in the slide calculates the average salary for each `department_id` and then displays the maximum average salary. Note that `GROUP BY` clause is mandatory when nesting group functions.
Summary

In this lesson, you should have learned how to:

• Use the group functions COUNT, MAX, MIN, SUM, and AVG
• Write queries that use the GROUP BY clause
• Write queries that use the HAVING clause

```
SELECT  column, group_function
FROM     table
[WHERE  condition]
[GROUP BY group_by_expression]
[HAVING  group_condition]
[ORDER BY column];
```

Summary

There are several group functions available in SQL, such as:

- AVG, COUNT, MAX, MIN, SUM, STDDEV, and VARIANCE

You can create subgroups by using the GROUP BY clause. Further, groups can be restricted using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. The order of the GROUP BY and HAVING clauses following the WHERE clause is not important. Place the ORDER BY clause at the end.

The Oracle server evaluates the clauses in the following order:

1. If the statement contains a WHERE clause, the server establishes the candidate rows.
2. The server identifies the groups that are specified in the GROUP BY clause.
3. The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

Note: For a complete list of the group functions, see Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Practice 5: Overview

This practice covers the following topics:
- Writing queries that use the group functions
- Grouping by rows to achieve more than one result
- Restricting groups by using the `HAVING` clause

Practice 5: Overview

At the end of this practice, you should be familiar with using group functions and selecting groups of data.
Practice 5

Determine the validity of the following three statements. Circle either True or False.

1. Group functions work across many rows to produce one result per group.
   True/False

2. Group functions include nulls in calculations.
   True/False

3. The WHERE clause restricts rows before inclusion in a group calculation.
   True/False

The HR department needs the following reports:

4. Find the highest, lowest, sum, and average salary of all employees. Label the columns as Maximum, Minimum, Sum, and Average, respectively. Round your results to the nearest whole number. Save your SQL statement as lab_05_04.sql. Run the query.

![Salary Data](image)

5. Modify the query in lab_05_04.sql to display the minimum, maximum, sum, and average salary for each job type. Resave lab_05_04.sql as lab_05_05.sql. Run the statement in lab_05_05.sql.
Practice 5 (continued)

6. Write a query to display the number of people with the same job.

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC_ACCOUNT</td>
<td>1</td>
</tr>
<tr>
<td>AC_MGR</td>
<td>1</td>
</tr>
<tr>
<td>AD_ASST</td>
<td>1</td>
</tr>
<tr>
<td>AD_PRES</td>
<td>1</td>
</tr>
<tr>
<td>AD_VP</td>
<td>2</td>
</tr>
<tr>
<td>IT_PROG</td>
<td>3</td>
</tr>
<tr>
<td>MK_MAN</td>
<td>1</td>
</tr>
<tr>
<td>MK_REP</td>
<td>1</td>
</tr>
<tr>
<td>SA_MAN</td>
<td>1</td>
</tr>
<tr>
<td>SA_REP</td>
<td>3</td>
</tr>
<tr>
<td>ST_CLERK</td>
<td>4</td>
</tr>
<tr>
<td>ST_MAN</td>
<td>1</td>
</tr>
</tbody>
</table>

Generalize the query so that the user in the HR department is prompted for a job title. Save the script to a file named `lab_05_06.sql`. Run the query. Enter `IT_PROG` when prompted.

7. Determine the number of managers without listing them. Label the column as `Number of Managers`. **Hint: Use the MANAGER_ID column to determine the number of managers.**

8. Find the difference between the highest and lowest salaries. Label the column `DIFFERENCE`. 
Practice 5 (continued)

If you have time, complete the following exercises:

9. Create a report to display the manager number and the salary of the lowest-paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is $6,000 or less. Sort the output in descending order of salary.

<table>
<thead>
<tr>
<th>MANAGER_ID</th>
<th>MIN(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>205</td>
</tr>
<tr>
<td>3</td>
<td>149</td>
</tr>
</tbody>
</table>

If you want an extra challenge, complete the following exercises:

10. Create a query to display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11. Create a matrix query to display the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

<table>
<thead>
<tr>
<th>Job</th>
<th>Dept 20</th>
<th>Dept 50</th>
<th>Dept 80</th>
<th>Dept 90</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT_PROG</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>19200</td>
</tr>
<tr>
<td>AC_MGR</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>12000</td>
</tr>
<tr>
<td>AC_ACCOUNT</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>8300</td>
</tr>
<tr>
<td>ST_MAN</td>
<td>(null)</td>
<td>5800</td>
<td>(null)</td>
<td>(null)</td>
<td>5800</td>
</tr>
<tr>
<td>AD_ASST</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>4400</td>
</tr>
<tr>
<td>AD_VP</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>34000</td>
<td>34000</td>
</tr>
<tr>
<td>SA_MAN</td>
<td>(null)</td>
<td>(null)</td>
<td>10500</td>
<td>(null)</td>
<td>10500</td>
</tr>
<tr>
<td>MK_MAN</td>
<td>13000</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>13000</td>
</tr>
<tr>
<td>AD_PRES</td>
<td>(null)</td>
<td>(null)</td>
<td>24000</td>
<td>(null)</td>
<td>24000</td>
</tr>
<tr>
<td>SA_REP</td>
<td>(null)</td>
<td>(null)</td>
<td>19600</td>
<td>(null)</td>
<td>26600</td>
</tr>
<tr>
<td>MK_REP</td>
<td>6000</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
<td>6000</td>
</tr>
<tr>
<td>ST_CLERK</td>
<td>(null)</td>
<td>11700</td>
<td>(null)</td>
<td>(null)</td>
<td>11700</td>
</tr>
</tbody>
</table>
Displaying Data from Multiple Tables
Objectives

After completing this lesson, you should be able to do the following:

• Write `SELECT` statements to access data from more than one table using equijoins and nonequijoins
• 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

Objectives

This lesson explains how to obtain data from more than one table. A `join` is used to view information from multiple tables. Therefore, you can `join` tables together to view information from more than one table.

Note: Information on joins is found in the section on `SQL Queries and Subqueries: Joins` in `Oracle Database SQL Language Reference 11g, Release 1 (11.1).`
Lesson Agenda

- **Types of JOINS and its syntax**
  - Natural join:
    - USING clause
    - ON clause
  - Self-join
  - Nonequijoins
  - OUTER join:
    - LEFT OUTER join
    - RIGHT OUTER join
    - FULL OUTER join
  - Cartesian product
    - Cross join
Obtaining Data from Multiple Tables

Sometimes you need to use data from more than one table. In the example in the slide, the report displays data from two separate tables:

- Employee IDs exist in the EMPLOYEES table.
- Department IDs exist in both the EMPLOYEES and DEPARTMENTS tables.
- Department names exist in the DEPARTMENTS table.

To produce the report, you need to link the EMPLOYEES and DEPARTMENTS tables, and access data from both of them.
Types of Joins

Joins that are compliant with the SQL:1999 standard include the following:

- **Natural joins:**
  - `NATURAL JOIN` clause
  - `USING` clause
  - `ON` clause

- **Outer joins:**
  - `LEFT OUTER JOIN`
  - `RIGHT OUTER JOIN`
  - `FULL OUTER JOIN`

- **Cross joins**

**Types of Joins**

To join tables, you can use a join syntax that is compliant with the SQL:1999 standard.

**Note:** Before the Oracle9i release, the join syntax was different from the American National Standards Institute (ANSI) standards. The SQL:1999–compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax that existed in the prior releases. For detailed information about the proprietary join syntax, see Appendix C: Oracle Join Syntax. **Note:** The following slide discusses the SQL:1999 join syntax.
Joining Tables Using SQL:1999 Syntax

Use a join to query data from more than one table:

```sql
SELECT table1.column, table2.column
FROM table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2 ON (table1.column_name = table2.column_name)] | 
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)]|
[CROSS JOIN table2];
```

Joining Tables Using SQL:1999 Syntax

In the syntax:
- `table1.column` denotes the table and the column from which data is retrieved
- `NATURAL JOIN` joins two tables based on the same column name
- `JOIN table2 USING column_name` performs an equijoin based on the column name
- `JOIN table2 ON table1.column_name = table2.column_name` performs an equijoin based on the condition in the `ON` clause
- `LEFT|RIGHT|FULL OUTER` is used to perform outer joins
- `CROSS JOIN` returns a Cartesian product from the two tables

For more information, see the section titled `SELECT` in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Instead of full table name prefixes, use table aliases.
- Table alias gives a table a shorter name:
  - Keeps SQL code smaller, uses less memory
- Use column aliases to distinguish columns that have identical names, but reside in different tables.

Qualifying Ambiguous Column Names

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT_ID column in the SELECT list could be from either the DEPARTMENTS table or the EMPLOYEES table. It is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

However, qualifying column names with table names can be time consuming, particularly if the table names are lengthy. Instead, you can use table aliases. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore using less memory.

The table name is specified in full, followed by a space and then the table alias. For example, the EMPLOYEES table can be given an alias of e, and the DEPARTMENTS table an alias of d.

Guidelines
- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid for only the current SELECT statement.
Lesson Agenda

- Types of `JOINS` and its syntax
  - Natural join:
    - `USING` clause
    - `ON` clause
  - Self-join
  - Nonequijoins
  - `OUTER` join:
    - `LEFT OUTER` join
    - `RIGHT OUTER` join
    - `FULL OUTER` join
  - Cartesian product
    - Cross join
Creating Natural Joins

- The NATURAL JOIN clause is based on all columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

Creating Natural Joins

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the NATURAL JOIN keywords.

**Note:** The join can happen on only those columns that have the same names and data types in both tables. If the columns have the same name but different data types, then the NATURAL JOIN syntax causes an error.
Retrieving Records with Natural Joins

In the example in the slide, the LOCATIONS table is joined to the DEPARTMENT table by the LOCATION_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

Natural Joins with a WHERE Clause
Additional restrictions on a natural join are implemented by using a WHERE clause. The following example limits the rows of output to those with a department ID equal to 20 or 50:

```
SELECT department_id, department_name, location_id, city
FROM departments
NATURAL JOIN locations
WHERE department_id IN (20, 50);
```
Creating Joins with the USING Clause

• If several columns have the same names but the data types do not match, natural join can be applied using the USING clause to specify the columns that should be used for an equijoin.

• Use the USING clause to match only one column when more than one column matches.

• The NATURAL JOIN and USING clauses are mutually exclusive.

Creating Joins with the USING Clause

Natural joins use all columns with matching names and data types to join the tables. The USING clause can be used to specify only those columns that should be used for an equijoin.
Joining Column Names

To determine an employee’s department name, you compare the value in the DEPARTMENT_ID column in the EMPLOYEES table with the DEPARTMENT_ID values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is an equijoin; that is, values in the DEPARTMENT_ID column in both the tables must be equal. Frequently, this type of join involves primary and foreign key complements.

Note: Equijoins are also called simple joins or inner joins.
Retrieving Records with the **USING** Clause

In the example in the slide, the `DEPARTMENT_ID` columns in the `EMPLOYEES` and `DEPARTMENTS` tables are joined and thus the `LOCATION_ID` of the department where an employee works is shown.
Using Table Aliases with the USING Clause

- Do not qualify a column that is used in the USING clause.
- If the same column is used elsewhere in the SQL statement, do not alias it.

```sql
SELECT l.city, d.department_name
FROM   locations l JOIN departments d
USING (location_id)
WHERE d.location_id = 1400;
```

Using Table Aliases with the USING clause

When joining with the USING clause, you cannot qualify a column that is used in the USING clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it. For example, in the query mentioned in the slide, you should not alias the location_id column in the WHERE clause because the column is used in the USING clause.

The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement. For example, the following statement is valid:

```
SELECT l.city, d.department_name
FROM   locations l JOIN departments d USING (location_id)
WHERE location_id = 1400;
```

Because, other columns that are common in both the tables, but not used in the USING clause, must be prefixed with a table alias otherwise you get the “column ambiguously defined” error.

In the following statement, manager_id is present in both the employees and departments table and if manager_id is not prefixed with a table alias, it gives a “column ambiguously defined” error.

The following statement is valid:

```
SELECT first_name, d.department_name, d.manager_id
FROM   employees e JOIN departments d USING (department_id)
WHERE department_id = 50;
```
Creating Joins with the ON Clause

• The join condition for the natural join is basically an equijoin of all columns with the same name.
• Use the ON clause to specify arbitrary conditions or specify columns to join.
• The join condition is separated from other search conditions.
• The ON clause makes code easy to understand.

Creating Joins with the ON Clause

Use the ON clause to specify a join condition. With this, you can specify join conditions separate from any search or filter conditions in the WHERE clause.
Retrieving Records with the **ON** Clause

In this example, the `DEPARTMENT_ID` columns in the `EMPLOYEES` and `DEPARTMENTS` table are joined using the **ON** clause. Wherever a department ID in the `EMPLOYEES` table equals a department ID in the `DEPARTMENTS` table, the row is returned. The table alias is necessary to qualify the matching column names.

You can also use the **ON** clause to join columns that have different names. The parenthesis around the joined columns as in the slide example, `(e.department_id = d.department_id)` is optional. So, even `ON e.department_id = d.department_id` will work.

**Note:** SQL Developer suffixes a ‘_1’ to differentiate between the two `department_ids`. 
Creating Three-Way Joins with the ON Clause

A three-way join is a join of three tables. In SQL:1999–compliant syntax, joins are performed from left to right. So, the first join to be performed is EMPLOYEES JOIN DEPARTMENTS. The first join condition can reference columns in EMPLOYEES and DEPARTMENTS but cannot reference columns in LOCATIONS. The second join condition can reference columns from all three tables.

**Note:** The code example in the slide can also be accomplished with the USING clause:

```sql
SELECT e.employee_id, l.city, d.department_name
FROM employees e
JOIN departments d
JOIN locations l
ON d.department_id = e.department_id
ON d.location_id = l.location_id;
```

---

```sql
SELECT e.employee_id, l.city, d.department_name
FROM employees e
JOIN departments d
USING (department_id)
JOIN locations l
USING (location_id)
```
Applying Additional Conditions to a Join

Use the **AND** clause or the **WHERE** clause to apply additional conditions:

```sql
SELECT e.employee_id, e.last_name, e.department_id, d.department_id, d.location_id
FROM employees e JOIN departments d
ON (e.department_id = d.department_id)
AND e.manager_id = 149;
```

Or

```sql
SELECT e.employee_id, e.last_name, e.department_id, d.department_id, d.location_id
FROM employees e JOIN departments d
ON (e.department_id = d.department_id)
WHERE e.manager_id = 149;
```

Applying Additional Conditions to a Join

You can apply additional conditions to the join.

The example shown performs a join on the `EMPLOYEES` and `DEPARTMENTS` tables and, in addition, displays only employees who have a manager ID of 149. To add additional conditions to the **ON** clause, you can add **AND** clauses. Alternatively, you can use a **WHERE** clause to apply additional conditions.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_ID_1</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>174 Abel</td>
<td>00</td>
<td>00</td>
<td>2500</td>
</tr>
<tr>
<td>2</td>
<td>176 Taylor</td>
<td>80</td>
<td>80</td>
<td>2500</td>
</tr>
</tbody>
</table>
Lesson Agenda

• Types of JOINS and its syntax
  • Natural join:
    – USING clause
    – ON clause
  • Self-join
  • Nonequijoins
  • OUTER join:
    – LEFT OUTER join
    – RIGHT OUTER join
    – FULL OUTER join
• Cartesian product
  – Cross join
Joining a Table to Itself

Sometimes you need to join a table to itself. To find the name of each employee’s manager, you need to join the EMPLOYEES table to itself, or perform a self-join. For example, to find the name of Lorentz’s manager, you need to:

- Find Lorentz in the EMPLOYEES table by looking at the LAST_NAME column.
- Find the manager number for Lorentz by looking at the MANAGER_ID column. Lorentz’s manager number is 103.
- Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold’s employee number is 103, so Hunold is Lorentz’s manager.

In this process, you look in the table twice. The first time you look in the table to find Lorentz in the LAST_NAME column and the MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.
Self-Joins Using the ON Clause

The ON clause can also be used to join columns that have different names, within the same table or in a different table.

The example shown is a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns.

**Note:** The parenthesis around the joined columns as in the slide example, `(e.manager_id = m.employee_id)` is optional. So, even `ON e.manager_id = m.employee_id` will work.
Lesson Agenda

• Types of JOINS and its syntax
• Natural join:
  – USING clause
  – ON clause
• Self-join
• Nonequijoin
• OUTER join:
  – LEFT OUTER join
  – RIGHT OUTER join
  – FULL OUTER join
• Cartesian product
  – Cross join
Nonequijoins

A nonequijoin is a join condition containing something other than an equality operator. The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a nonequijoin. The SALARY column in the EMPLOYEES table ranges between the values in the LOWEST_SAL and HIGHEST_SAL columns of the JOB_GRADES table. Hence, each employee can be graded based on their salary. The relationship is obtained using an operator other than the equality (=) operator.
Retrieving Records with Nonequijoins

The slide example creates a nonequijoin to evaluate an employee’s salary grade. The salary must be between any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the JOB_GRADES table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees’ salaries lie within the limits provided by the job grade table. That is, no employee earns less than the lowest value contained in the LOWEST_SAL column or more than the highest value contained in the HIGHEST_SAL column.

**Note:** Other conditions (such as <= and >=) can be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using the BETWEEN condition. The Oracle server translates the BETWEEN condition to a pair of AND conditions. Therefore, using BETWEEN has no performance benefits, but should be used only for logical simplicity.

Table aliases have been specified in the slide example for performance reasons, not because of possible ambiguity.
Lesson Agenda

- Types of JOINs and its syntax
- Natural join:
  - USING clause
  - ON clause
- Self-join
- Nonequijoins
- OUTER join:
  - LEFT OUTER join
  - RIGHT OUTER join
  - FULL OUTER join
- Cartesian product
  - Cross join
Returning Records with No Direct Match with Outer Joins

If a row does not satisfy a join condition, the row does not appear in the query result. For example, in the equijoin condition of EMPLOYEES and DEPARTMENTS tables, department ID 190 does not appear because there are no employees with that department ID recorded in the EMPLOYEES table. Therefore, instead of seeing 20 employees in the result set, you see 19 records.

To return the department record that does not have any employees, you can use an outer join.
INNER Versus OUTER Joins

• In SQL:1999, the join of two tables returning only matched rows is called an inner join.

• A join between two tables that returns the results of the inner join as well as the unmatched rows from the left (or right) table is called a left (or right) outer join.

• A join between two tables that returns the results of an inner join as well as the results of a left and right join is a full outer join.

INNER Versus OUTER Joins

Joining tables with the NATURAL JOIN, USING, or ON clauses results in an inner join. Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an outer join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of outer joins:

• LEFT OUTER
• RIGHT OUTER
• FULL OUTER
LEFT OUTER JOIN

This query retrieves all rows in the EMPLOYEES table, which is the left table, even if there is no match in the DEPARTMENTS table.
RIGHT OUTER JOIN

This query retrieves all rows in the `DEPARTMENTS` table, which is the right table, even if there is no match in the `EMPLOYEES` table.
**FULL OUTER JOIN**

This query retrieves all rows in the `EMPLOYEES` table, even if there is no match in the `DEPARTMENTS` table. It also retrieves all rows in the `DEPARTMENTS` table, even if there is no match in the `EMPLOYEES` table.

```sql
SELECT e.last_name, d.department_id, d.department_name
FROM employees e FULL OUTER JOIN departments d
ON (e.department_id = d.department_id);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whitman</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td><strong>...</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taylor</td>
<td>80</td>
<td>Sales</td>
</tr>
<tr>
<td>Grant</td>
<td>(null)</td>
<td>(null)</td>
</tr>
<tr>
<td>(null)</td>
<td>150</td>
<td>Contracting</td>
</tr>
</tbody>
</table>
Lesson Agenda

• Types of JOINS and its syntax
• Natural join:
  – USING clause
  – ON clause
• Self-join
• Nonequijoin
• OUTER join:
  – LEFT OUTER join
  – RIGHT OUTER join
  – FULL OUTER join
• Cartesian product
  – Cross join
Cartesian Products

• A Cartesian product is formed when:
  – A join condition is omitted
  – A join condition is invalid
  – All rows in the first table are joined to all rows in the second table

• To avoid a Cartesian product, always include a valid join condition.

Cartesian Products

When a join condition is invalid or omitted completely, the result is a Cartesian product, in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table.

A Cartesian product tends to generate a large number of rows and the result is rarely useful. You should, therefore, always include a valid join condition unless you have a specific need to combine all rows from all tables.

However, Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.
Generating a Cartesian Product

A Cartesian product is generated if a join condition is omitted. The example in the slide displays the employee last name and the department name from the `EMPLOYEES` and `DEPARTMENTS` tables. Because no join condition was specified, all rows (20 rows) from the `EMPLOYEES` table are joined with all rows (8 rows) in the `DEPARTMENTS` table, thereby generating 160 rows in the output.
Creating Cross Joins

- The **CROSS JOIN** clause produces the cross-product of two tables.
- This is also called a Cartesian product between the two tables.

```sql
SELECT last_name, department_name
FROM employees
CROSS JOIN departments;
```

<table>
<thead>
<tr>
<th>#</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abel</td>
<td>Administration</td>
</tr>
<tr>
<td>2</td>
<td>Davis</td>
<td>Administration</td>
</tr>
<tr>
<td>3</td>
<td>DeHasan</td>
<td>Administration</td>
</tr>
<tr>
<td>4</td>
<td>Ernst</td>
<td>Administration</td>
</tr>
<tr>
<td>5</td>
<td>Pay</td>
<td>Administration</td>
</tr>
</tbody>
</table>

***

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Contracting</th>
</tr>
</thead>
<tbody>
<tr>
<td>159</td>
<td>Whalen</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>Zidley</td>
<td></td>
</tr>
</tbody>
</table>

Creating Cross Joins

The example in the slide produces a Cartesian product of the `EMPLOYEES` and `DEPARTMENTS` tables.
Summary

In this lesson, you should have learned how to use joins to display data from multiple tables by using:

- Equijoins
- Nonequijoins
- Outer joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) outer joins

Summary

There are multiple ways to join tables.

Types of Joins

- Equijoins
- Nonequijoins
- Outer joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) outer joins

Cartesian Products

A Cartesian product results in the display of all combinations of rows. This is done by either omitting the WHERE clause or by specifying the CROSS JOIN clause.

Table Aliases

- Table aliases speed up database access.
- Table aliases can help to keep SQL code smaller by conserving memory.
- Table aliases are sometimes mandatory to avoid column ambiguity.
Practice 6: Overview

This practice covers the following topics:

• Joining tables using an equijoin
• Performing outer and self-joins
• Adding conditions

Practice 6: Overview

This practice is intended to give you experience in extracting data from more than one table using the SQL:1999–compliant joins.
Practice 6

1. Write a query for the HR department to produce the addresses of all the departments. Use the LOCATIONS and COUNTRIES tables. Show the location ID, street address, city, state or province, and country in the output. Use a NATURAL JOIN to produce the results.

<table>
<thead>
<tr>
<th>LOCATION_ID</th>
<th>STREET_ADDRESS</th>
<th>CITY</th>
<th>STATE_PROVINCE</th>
<th>COUNTRY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1400 2014 Jabberwocky Rd</td>
<td>Southlake</td>
<td>Texas</td>
<td>United States of America</td>
</tr>
<tr>
<td>2</td>
<td>1500 2011 Interiors Blvd</td>
<td>South San Francisco</td>
<td>California</td>
<td>United States of America</td>
</tr>
<tr>
<td>3</td>
<td>1700 2004 Charade Rd</td>
<td>Seattle</td>
<td>Washington</td>
<td>United States of America</td>
</tr>
<tr>
<td>4</td>
<td>1800 460 Bloor St. W.</td>
<td>Toronto</td>
<td>Ontario</td>
<td>Canada</td>
</tr>
<tr>
<td>5</td>
<td>2500 Magdalen Centre, The ...</td>
<td>Oxford</td>
<td>Oxford</td>
<td>United Kingdom</td>
</tr>
</tbody>
</table>

2. The HR department needs a report of all employees. Write a query to display the last name, department number, and department name for all the employees.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Davies</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Vargas</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Mourgos</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Matos</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Hunold</td>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td>Ernst</td>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Getz</td>
<td>110</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

...
Practice 6 (continued)

3. The HR department needs a report of employees in Toronto. Display the last name, job, department number, and the department name for all employees who work in Toronto.

<table>
<thead>
<tr>
<th>Last_Name</th>
<th>Job_ID</th>
<th>Department_ID</th>
<th>Department_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>MK_REP</td>
<td>20</td>
<td>Marketing</td>
</tr>
</tbody>
</table>

4. Create a report to display employees’ last name and employee number along with their manager’s last name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively. Save your SQL statement as lab_06_04.sql. Run the query.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Emp#</th>
<th>Manager</th>
<th>Mgr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
<td>101</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>De Haan</td>
<td>102</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Hunold</td>
<td>103</td>
<td>De Haan</td>
<td>102</td>
</tr>
<tr>
<td>Ernst</td>
<td>104</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Lorentz</td>
<td>107</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Mourgos</td>
<td>124</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Rajs</td>
<td>141</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Davies</td>
<td>142</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Matos</td>
<td>143</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Vargas</td>
<td>144</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Whalen</td>
<td>200</td>
<td>Kochhar</td>
<td>101</td>
</tr>
<tr>
<td>Hartstein</td>
<td>201</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Fay</td>
<td>202</td>
<td>Hartstein</td>
<td>201</td>
</tr>
<tr>
<td>Higgins</td>
<td>205</td>
<td>Kochhar</td>
<td>101</td>
</tr>
<tr>
<td>Gietz</td>
<td>206</td>
<td>Higgins</td>
<td>205</td>
</tr>
</tbody>
</table>
Practice 6 (continued)

5. Modify `lab_06_04.sql` to display all employees including King, who has no manager. Order the results by the employee number. Save your SQL statement as `lab_06_05.sql`. Run the query in `lab_06_05.sql`.

<table>
<thead>
<tr>
<th>Employee</th>
<th>EMP#</th>
<th>Manager</th>
<th>Mgr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 King</td>
<td>100</td>
<td>(null)</td>
<td>(null)</td>
</tr>
<tr>
<td>2 Kochhar</td>
<td>101</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>3 De Haan</td>
<td>102</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>4 Hunold</td>
<td>103</td>
<td>De Haan</td>
<td>102</td>
</tr>
<tr>
<td>5 Ernst</td>
<td>104</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>6 Lorentz</td>
<td>107</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>7 Mourgos</td>
<td>124</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>8 Rajs</td>
<td>141</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>9 Davies</td>
<td>142</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>10 Matos</td>
<td>143</td>
<td>Mourgos</td>
<td>124</td>
</tr>
</tbody>
</table>

6. Create a report for the HR department that displays employee last names, department numbers, and all the employees who work in the same department as a given employee. Give each column an appropriate label. Save the script to a file named `lab_06_06.sql`.

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>EMPLOYEE</th>
<th>COLLEAGUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20 Fay</td>
<td>Hartstein</td>
</tr>
<tr>
<td>2</td>
<td>20 Hartstein</td>
<td>Fay</td>
</tr>
<tr>
<td>3</td>
<td>50 Davies</td>
<td>Matos</td>
</tr>
<tr>
<td>4</td>
<td>50 Davies</td>
<td>Mourgos</td>
</tr>
<tr>
<td>5</td>
<td>50 Davies</td>
<td>Rajs</td>
</tr>
<tr>
<td>6</td>
<td>50 Davies</td>
<td>Vargas</td>
</tr>
<tr>
<td>7</td>
<td>50 Matos</td>
<td>Davies</td>
</tr>
<tr>
<td>8</td>
<td>50 Matos</td>
<td>Mourgos</td>
</tr>
<tr>
<td>9</td>
<td>50 Matos</td>
<td>Rajs</td>
</tr>
<tr>
<td>10</td>
<td>50 Matos</td>
<td>Vargas</td>
</tr>
<tr>
<td>42</td>
<td>110 Higgins</td>
<td>Gietz</td>
</tr>
</tbody>
</table>
Practice 6 (continued)

7. The HR department needs a report on job grades and salaries. To familiarize yourself with the `JOB_GRADES` table, first show the structure of the `JOB_GRADES` table. Then create a query that displays the name, job, department name, salary, and grade for all employees.

```
DESC JOB_GRADES
Name Null Type
--------------------------
GRADE_LEVEL VARCHAR2(3)
LOWEST_SAL NUMBER
HIGHEST_SAL NUMBER

3 rows selected
```

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_NAME</th>
<th>SALARY</th>
<th>GRADE_LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>2500 A</td>
<td>A</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>2600 A</td>
<td>A</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>3100 B</td>
<td>B</td>
</tr>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>3500 B</td>
<td>B</td>
</tr>
<tr>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>IT</td>
<td>4200 B</td>
<td>B</td>
</tr>
<tr>
<td>Whalen</td>
<td>AD_ASST</td>
<td>Administration</td>
<td>4400 B</td>
<td>B</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>Shipping</td>
<td>5800 B</td>
<td>B</td>
</tr>
<tr>
<td>Ernst</td>
<td>IT_PROG</td>
<td>IT</td>
<td>6000 C</td>
<td>C</td>
</tr>
<tr>
<td>Fay</td>
<td>MK_REP</td>
<td>Marketing</td>
<td>6000 C</td>
<td>C</td>
</tr>
<tr>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>Accounting</td>
<td>8300 C</td>
<td>C</td>
</tr>
<tr>
<td>De Haan</td>
<td>AD_VP</td>
<td>Executive</td>
<td>17000 E</td>
<td></td>
</tr>
<tr>
<td>King</td>
<td>AD_PRES</td>
<td>Executive</td>
<td>24000 E</td>
<td></td>
</tr>
</tbody>
</table>
```
Practice 6 (continued)
If you want an extra challenge, complete the following exercises:

8. The HR department wants to determine the names of all the employees who were hired after Davies. Create a query to display the name and hire date of any employee hired after employee Davies.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lorentz</td>
<td>07-FEB-99</td>
</tr>
<tr>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Matos</td>
<td>15-MAR-98</td>
</tr>
<tr>
<td>Vargas</td>
<td>09-JUL-98</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Taylor</td>
<td>24-MAR-98</td>
</tr>
<tr>
<td>Grant</td>
<td>24-MAY-99</td>
</tr>
<tr>
<td>Fay</td>
<td>17-AUG-97</td>
</tr>
</tbody>
</table>

9. The HR department needs to find the names and hire dates of all the employees who were hired before their managers, along with their managers’ names and hire dates. Save the script to a file named lab_06_09.sql.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>LAST_NAME_1</th>
<th>HIRE_DATE_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>17-SEP-87</td>
<td>Kochhar</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>Hunold</td>
<td>03-JAN-90</td>
<td>De Haan</td>
<td>13-JAN-93</td>
</tr>
<tr>
<td>Vargas</td>
<td>09-JUL-98</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Matos</td>
<td>15-MAR-98</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Davies</td>
<td>29-JAN-97</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Rajs</td>
<td>17-OCT-95</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Grant</td>
<td>24-MAY-99</td>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Taylor</td>
<td>24-MAR-98</td>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Abel</td>
<td>11-MAY-96</td>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
</tbody>
</table>