CONTENTS

Chapter 1 - Course Introduction

Course Objectives

- ❈ Describe the features of a Relational Database.
- ❈ Interact with a Relational Database Management System.
- ❈ Use SQL*Plus to connect to an Oracle database and submit SQL statements.
- ❈ Write SQL queries.
- ❈ Use SQL functions.
- ❈ Use a query to join together data items from multiple tables.
- ❈ Write nested queries.
- ❈ Perform summary analysis of data in a query.
- ❈ Add, change, and remove data in a database.
- ❈ Manage database transactions.
- ❈ Work in a multi-user database environment.
- ❈ Create and manage tables and other database objects.
- ❈ Control access to data.

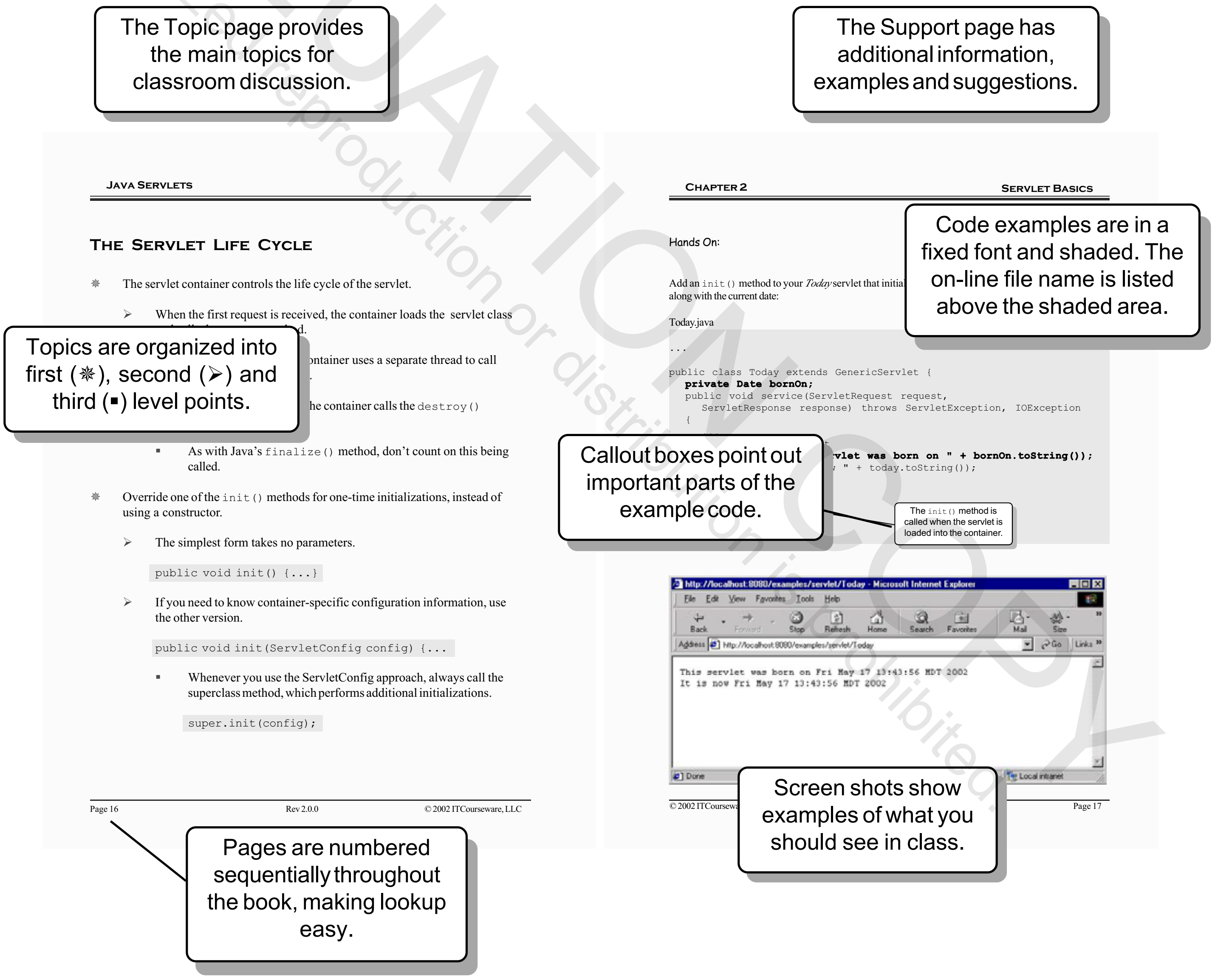
COURSE OVERVIEW

- **Audience:** This course is designed for database application developers.
- **Prerequisites:** Familiarity with relational database concepts.
- **Student Materials:**
  - Student workbook
- **Classroom Environment:**
  - One workstation per student
  - Oracle 12c

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.



The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

❋ The servlet container controls the life cycle of the servlet.

➢ When the first request is received, the container loads the servlet class

ontainer uses a separate thread to call

he container calls the `destroy()`

▪ As with Java's `finalize()` method, don't count on this being called.

❋ Override one of the `init()` methods for one-time initializations, instead of using a constructor.

➢ The simplest form takes no parameters.

```
public void init() {...}
```

➢ If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) {...
```

▪ Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

Topics are organized into first (❋), second (➢) and third (▪) level points.

Pages are numbered sequentially throughout the book, making lookup easy.

CHAPTER 2 SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initial along with the current date:

Today.java

```
...
public class Today extends GenericServlet {
   private Date bornOn;
   public void service(ServletRequest request,
      ServletResponse response) throws ServletException, IOException
   {
                   vlet was born on " + bornOn.toString());
                   " + today.toString());
```

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

The `init()` method is called when the servlet is loaded into the container.

http://localhost:8080/examples/servlet/Today - Microsoft Internet Explorer

```
This servlet was born on Fri May 17 13:43:56 MDT 2002
It is now Fri May 17 13:43:56 MDT 2002
```

Screen shots show examples of what you should see in class.

© 2002 ITCoursew Page 17

Suggested References

Beaulieu, Alan. 2009. ***Learning SQL***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596520823

Celko, Joe. 2010. ***Joe Celko's SQL for Smarties, Fourth Edition: Advanced SQL Programming***. Academic Press/ Morgan Kaufman, San Francisco, CA. ISBN 0123820227

Celko, Joe. 2006. ***Joe Celko's SQL Puzzles and Answers***. Morgan Kaufmann, San Francisco,CA. ISBN 0123735963

Churcher, Clare. 2008. ***Beginning SQL Queries: From Novice to Professional***. Apress, Inc., Berkeley, CA. ISBN 9781590599433

Date, C.J. and Hugh Darwen. 1996. ***A Guide to The SQL Standard, Fourth Edition***. Addison-Wesley, Reading, MA. ISBN 0201964260

Date, C. J.. 2011. ***SQL and Relational Theory, Second Edition: How to Write Accurate SQL Code***. O'Reilly Media, Sebastopol, CA. ISBN 1449316409

Gennick, Jonathan. 2004. ***Oracle Sql*Plus Pocket Reference, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596008856

Gruber, Martin. 2000. ***SQL Instant Reference, Second Edition***. SYBEX, Alameda, CA. ISBN 0782125395

Kline, Kevin. 2009. ***SQL in a Nutshell, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596518847

Kreines, David. 2003. ***Oracle Data Dictionary Pocket Reference***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005172

Loney, Kevin. 2013. ***Oracle Database 12c: The Complete Reference***. McGraw-Hill Osborne Media, ISBN 0071801758

Mishra, Sanjay. 2009. ***Mastering Oracle SQL, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596006322

http://docs.oracle.com
http://www.dbasupport.com
http://www.orafaq.com
http://asktom.oracle.com
http://www.oracle.com
http://www.searchdatabase.com

Your single most important reference is the ***SQL Language Reference*** book, which is part of the *Oracle Database Online Documentation Library*. This is the official, complete description of Oracle's implementation of SQL. It includes many examples and discussions.

An easy way to find it is to go to:

http://docs.oracle.com/

Find the documentation for your version of Oracle Database. Locate the *SQL Language Reference*, and open the HTML table of contents.

If you have web access in the classroom, open a browser now and find the *SQL Language Reference*. Set a browser bookmark, and have the *SQL Language Reference* at hand throughout this class.

CHAPTER 2 - RELATIONAL DATABASE AND SQL OVERVIEW

OBJECTIVES

- ❈ Describe the features of a Relational Database.
- ❈ Describe the features of a Relational Database Management System.
- ❈ Describe the standard SQL datatypes.

REVIEW OF RELATIONAL DATABASE TERMINOLOGY

- A *Relational Database* consists of *tables*, each with a specific name.

- A table is organized in *columns*, each with a specific name and each capable of storing a specific *datatype*.

- A *row* is a distinct set of values, one value for each column (although a column value might be empty (*null*) for a particular row).

- Each table can have a *primary key*, consisting of one or more columns.

  - The set of values in the primary key column or columns must be unique and not null for each row.

- One table might contain a column or columns that correspond to the primary key or unique key of another table; this is called a *foreign key*.

- A *view*, or *result set*, is a collection of row and column values derived by querying tables.

A *Relational Database* (RDB) is a database which conforms to Foundation Rules defined by Dr. E. F. Codd. It is a particular method of organizing information.

A *Relational Database Management System* (RDBMS) is a software system that allows you to create and manage a Relational Database. Minimum requirements for such a system are defined by both ANSI and ISO. The most recent standard is named *SQL2*, since most of the standard simply defines the language (SQL) used to create and manage such a database and its data. Some people use the term *SQL Database* as a synonym for *Relational Database*.

Each row (sometimes called a *record*) represents a single entity in the real world. Each column (sometimes called a *field*) in that row represents an attribute of that entity.

Entity Relationship Modeling is the process of deciding which attributes of which entities you will store in your database, and how different entities are related to one another.

The formal word for row is *tuple*; that is, each row in a table that has three columns might be called a *triple* (a set of three attribute values); five columns, a *quintuple*; eight columns, an *octuple*; or, in general, however many attributes describe an entity of some sort, the set of column values in a row that represents one such entity is a tuple. The formal word for column is *attribute*.

RELATIONAL DATABASE MANAGEMENT SYSTEMS

- A *Relational Database Management System* (RDBMS) provides for users.
  - Each *user* is identified by an account name.
    - A user can access data and create database objects based on privileges granted by the database administrator.
  - Users own the tables they create; the set of tables (and other database objects) owned by a user is called a *schema*.
    - Users can grant *privileges* so that other users can access the schema.
- A *session* starts when you connect to the system.
- Once you connect to the database system, all your changes are considered a single *transaction* until you either *commit* or *rollback* your work.
- *SQL* is a standard language for querying, manipulating data, and creating and managing objects in your schema.
- The *Data Dictionary* (also called a *System Catalog* or *Online Catalog*) is a set of ordinary relational tables, maintained by the system, whose rows describe the tables and other items in your schema.
  - This is referred to as *metadata*.
  - You can query a system catalog table just like any other table.
  - Each database vendor has its own system catalog.

Each database product organizes their online catalog or data dictionary as they see fit. For example:

MySQL - the *mysql* database:

```
columns_priv              db
event                     func
general_log               help_category
help_keyword              help_relation
help_topic                host
ndb_binlog_index          plugin
proc                      procs_priv
servers                   slow_log
tables_priv               time_zone
time_zone_leap_second     time_zone_name
time_zone_transition      time_zone_transition_type
user
```

Microsoft SQL Server - Database System Tables (there are many other system tables):

```
syscolumns                sysindexkeys
syscomments               sysmembers
sysconstraints            sysobjects
sysdepends                syspermissions
sysfilegroups             sysprotects
sysfiles                  sysreferences
sysforeignkeys            systypes
sysfulltextcatalogs       sysusers
sysindexes
```

Oracle Database - Static Data Dictionary Views (there are hundreds of others, these are some commonly-used examples):

```
user_tables               all_tables
dba_tables                user_tab_columns
user_views                user_indexes
user_constraints          user_cons_columns
user_users                all_users
user_catalog              user_objects
user_tab_privs            user_col_privs
user_role_privs           user_sys_privs
user_tablespaces          user_ts_quotas
user_segments             user_triggers
database_properties       nls_session_parameters
recyclebin                dictionary
```

SQL Datatypes

- Each column in a table has a specific, predefined datatype.
  - Only values of the correct type can be stored in the column.
  - Many datatype definitions also include a limit on the size of the values that are allowed.
- The details of how data values are stored vary among database vendors.
  - Most database vendors provide similar sets of datatypes, though.
- The most important ANSI/ISO standard datatypes are:
  - **CHARACTER** — Text values of a specific predefined length; the database pads shorter values to the correct length with spaces.
  - **CHARACTER VARYING** — Text values whose length can vary, up to a predefined maximum length for each column.
  - **BIT, BIT VARYING** — Fixed or varying-length bit strings for binary data such as images or documents.
  - **INTEGER** — Whole number values, possibly with predefined precision.
  - **FLOAT, DOUBLE PRECISION FLOAT**— Floating-point numbers.
  - **NUMERIC/DECIMAL** — Fixed-point numbers, with predefined precision and scale.
  - **DATE, TIME, DATETIME, TIMESTAMP** — Calendar date, clock time, combined date/time, date/time with predefined precision.
  - **INTERVAL** — Time interval, perhaps with predefined precision.

While most SQL database vendors provide all the ANSI/ISO standard datatypes in one form or another, they vary widely in their implementation and internal storage format. Examples:

INTEGER:
* MySQL has **TINYINT** (1 byte, signed or unsigned), **SMALLINT** (2 bytes, signed or unsigned), **INT** or **INTEGER** (4 bytes, signed or unsigned), and **BIGINT** (8 bytes, signed or unsigned.)

* Microsoft SQL Server has **TINYINT** (1 byte, unsigned), **SMALLINT** (2 bytes, signed), **INT** (4 bytes, signed), and **BIGINT** (8 bytes, signed).

* Oracle Database has **NUMBER(*p*)**, where *p* specifies precision as the number of decimal digits (up to 38); values stored in a varying length proprietary binary format, consuming only as many bytes as are needed for each number.

CHARACTER VARYING:
* MySQL has **VARCHAR(*n*)** where *n* is the maximum number of characters allowed, up to 65,536.

* Microsoft SQL Server has **VARCHAR(*n*)**, where *n* is either the maximum number of bytes allowed, up to 8,000; or the word **max** indicating strings up to 2GB are allowed.

* Oracle Database has **VARCHAR2(*n*)**, where *n* is the maximum number of bytes allowed, up to 4,000. Starting in Oracle version 12c, this 4,000 byte limit can be increased by the database administrator to 32,767, though the default is still 4,000.

DATE, TIME, DATETIME:
* MySQL has **DATE** (calendar date, from 1000-01-01 ad to 9999-12-31 A.D.), **TIME** (clock time), and **DATETIME** (combined date and clock time), all stored in compact binary formats.

* Microsoft SQL Server has **DATE** (calendar date, from 0001-01-01 to 9999-12-31 A.D.), **TIME** (clock time), **DATETIME** (combined date and time from 1753-01-01 to 9999-12-31 A.D), **DATETIME2** (combined date and time from 0001-01-01 to 9999-12-31 A.D), and **SMALLDATETIME** (combined date and time from 1900-01-01 to 2079-06-06 A.D).

* Oracle Database has **DATE** (combined date and time from 4712-01-01 B.C to 9999-12-31 A.D.), stored in a compact 7-byte binary format.

In addition, each database product has its own rules for presenting and accepting dates as human-readable strings.

INTRODUCTION TO SQL

- SQL is the abbreviation for *Structured Query Language*.
  - It is often pronounced as "sequel."
- SQL was first developed by IBM in the mid-1970s.
- SQL is the international standard language for relational database management systems.
  - SQL is considered a fourth-generation language.
  - It is English-like and intuitive.
  - SQL is robust enough to be used by:
    - Users with non-technical backgrounds.
    - Professional developers.
    - Database administrators.
- SQL is a non-procedural language that emphasizes what to get, but not how to get it.
- Each vendor has its own implementation of SQL; most large vendors comply with the more recent ANSI/ISO standards, and may include proprietary language extensions for greater functionality.

SQL statements can be placed in two main categories:

Data Manipulation:

Query:	**SELECT**

Data Manipulation Language (DML):	**INSERT**
	UPDATE
	DELETE

Transaction Control:	**COMMIT** **ROLLBACK**

Data Definition:

Data Definition Language (DDL):	**CREATE** **ALTER** **DROP**
Data Control Language (DCL):	**GRANT** **REVOKE**

SQL is actually an easy language to learn (many users pick up the basics with no additional instruction). SQL statements look more like natural language than many other programming languages. We can parse them into "verbs," "clauses," and "predicates." Additionally, SQL is a compact language, making it easy to learn and remember. Users and programmers spend most of their time working with only four simple keywords (the Query and DML verbs in the list above). Of course, as we'll learn in this class, you can use them in sophisticated ways.

CHAPTER 7 - AGGREGATE FUNCTIONS AND ADVANCED TECHNIQUES

OBJECTIVES

- ❈ Describe and use regular and correlated subqueries.
- ❈ Use the **EXISTS** operator.
- ❈ Use aggregate functions to generate a summary row.
- ❈ Group summary rows by key values.
- ❈ Combine multiple queries using set operators.

SUBQUERIES

- A *subquery* is a **SELECT** that appears as part of another SQL statement.
  - In the **WHERE** clause of another **SELECT** statement - this is called a *nested subquery.*
  - In the **FROM** clause of another **SELECT** statement - this is called an *inline view*.
    - The result of the subquery is treated as though it were another table; you can even give the subquery a table alias.
  - As an expression in a **SELECT**, **VALUES**, or **SET** clause.
- A subquery must return the correct number and type of columns and rows.
  - A single-row subquery returns exactly one record, while a multi-row subquery might return more than one record.
  - Operators such as **=** and **!=** must be given a *scalar subquery*, a single-row subquery with just one column.

    subquery1.sql
    ```
    SELECT lastname, firstname, area_code, phone_number
      FROM person
     WHERE id = (SELECT manager_id
                   FROM store
                  WHERE store_number = 7);
    ```
  - Operators such as **IN** allow a multi-row subquery, one that can return any number of rows.

    subquery2.sql
    ```
    SELECT lastname, firstname, city, state
      FROM person
     WHERE state IN (SELECT state FROM store);
    ```

Report all personal orders made by the manager of Store #4:

invoice.sql
```
SELECT invoice_number, amount_due
  FROM order_header
 WHERE customer_id = (SELECT manager_id
                        FROM store
                       WHERE store_number = 4);
```

Show the name of the customer on Invoice #2345:

invoice2.sql
```
SELECT lastname, firstname
  FROM person
 WHERE id = (SELECT customer_id
               FROM order_header
              WHERE invoice_number = 2345);
```

When do you use a subquery versus a **JOIN** in a **SELECT** statement?

When the query is only returning column values from a single table, you can use a subquery in the **WHERE** clause to help limit the result set.

person_sub.sql
```
SELECT lastname, firstname
  FROM person
 WHERE id IN (SELECT id
                FROM employee);
```

When the query is returning column values from multiple tables, you will have to use a **JOIN** so that all of the table data is available in the main query.

person_join.sql
```
SELECT lastname, firstname, title
  FROM person JOIN employee USING (id);
```

CORRELATED SUBQUERIES

- A subquery whose **WHERE** clause refers to a table in the **FROM** clause of a parent query is a *correlated subquery*.
  - The subquery is correlated to specific values in each row processed by the parent query.

  corr_subquery.sql

```
SELECT lastname, firstname, invoice_number
  FROM person JOIN order_header oh ON id = customer_id
 WHERE customer_id = (SELECT manager_id
                        FROM store
                       WHERE store_number = oh.store_number);
```

  - Subqueries often can be rewritten as joins in **SELECT** statements.

- Oracle has an internal query optimizer.
  - Using a join, rather than a correlated subquery, gives Oracle the flexibility to determine the most efficient query technique.

    join.sql

```
SELECT lastname, firstname, invoice_number
  FROM person JOIN order_header oh ON id = customer_id
              JOIN store s ON customer_id = manager_id
                         AND s.store_number =
                             oh.store_number;
```

THE EXISTS OPERATOR

- The **EXISTS** operator is used in the **WHERE** clause of a correlated subquery to test for the existence of data; it does not return data.
  - The outer query relies on the subquery's boolean result to determine whether to include its current row in the resultset.

    ```
    WHERE [NOT] EXISTS (subquery);
    ```

- An **EXISTS** condition is true if its subquery returns at least one row.
- The item in the subquery's **SELECT** list is often the primary key or a *****.

  exists.sql

  ```
  SELECT lastname, firstname, store_number
    FROM person JOIN store ON id = manager_id
   WHERE EXISTS (SELECT invoice_number
                   FROM order_header oh
                  WHERE oh.customer_id = store.manager_id);
  ```

- The **EXISTS** operator results in a *semi-join*.
  - A matched row from the containing query appears only once, no matter how many matching rows would be found by the subquery.

There is always more than one way to do it in SQL:

not_exists.sql
```
SELECT product_id, description
  FROM product
 WHERE NOT EXISTS (SELECT *
                     FROM inventory
                    WHERE product_id = product.product_id
                      AND store_number = 7);
```

not_in.sql
```
SELECT product_id, description
  FROM product
 WHERE product_id NOT IN (SELECT product_id
                            FROM inventory
                           WHERE store_number = 7);
```

inline_view.sql
```
SELECT p.product_id, p.description
  FROM product p LEFT OUTER JOIN (SELECT product_id FROM inventory
                                   WHERE store_number = 7) i
                 ON p.product_id = i.product_id
 WHERE i.product_id IS NULL;
```

These are examples of *antijoins* — queries that return rows from one table for which there are no matching rows in the other table.

Although there is always more than one way to do it, not everything you try will actually work. Consider:

bad_join.sql
```
SELECT p.product_id, description
  FROM product p JOIN inventory i
                   ON p.product_id != i.product_id;
```

Exactly which rows will be selected by this join condition? (Hint: do NOT try running it!)
This results not in an antijoin, but in a Cartesian or **CROSS** join of **product** with **inventory**, with just the matching rows left out.

THE AGGREGATE FUNCTIONS

- The *aggregate* functions take a group of rows generated by a **SELECT** statement and calculate a single value.
- By default, aggregate functions will generate a single summary row for all values retrieved by the query.
- The most commonly used aggregate functions are:
  - **COUNT(*)** — Total count of all rows selected.
  - **COUNT(*column*)** — Total count of rows selected in which ***column*** is not **NULL**.
  - **SUM(*column*)** — Sum of values of ***column*** for selected rows.
  - **AVG(*column*)** — Average value of ***column*** for selected rows.
  - **MAX(*column*)** — Maximum value of ***column*** for selected rows.
  - **MIN(*column*)** — Minimum value of ***column*** for selected rows.
- The *column* may also be an expression that yields a value.

  inv_sum.sql
```
SELECT SUM(quantity_on_hand * price)
  FROM inventory JOIN product USING (product_id)
 WHERE store_number = 7;
```
- Aggregate functions are sometimes called *grouping*, *column*, or *set* functions.
- Aggregate functions cannot be used in a **WHERE** clause.
  - However, you can place the aggregate in a subquery's **SELECT** list.

    sub_agg.sql
```
SELECT id, title, pay_amount
  FROM employee WHERE pay_type_code = 'S'
   AND pay_amount < (SELECT MAX(pay_amount) * .5
                         FROM employee
                      WHERE pay_type_code = 'S');
```

order_count.sql

```
SELECT COUNT(*) FROM order_header;
```

avg_pay.sql

```
SELECT AVG(pay_amount) FROM employee;
```

min_pay.sql

```
SELECT MIN(pay_amount) FROM employee
 WHERE pay_type_code = 'H';
```

max_invoice.sql

```
SELECT MAX(invoice_number) FROM order_header;
```

Multiple aggregates can be used in a single query.

mult_agg.sql

```
SELECT MIN(pay_amount), AVG(pay_amount), MAX(pay_amount)
  FROM employee
 WHERE pay_type_code = 'S';
```

NULLS AND DISTINCT

- When you aggregate the values of a specific column, rows with null values for that column are skipped.

  account_orders.sql

  ```
  SELECT COUNT(account_number) AS order_on_acct
    FROM order_header;
  ```

  - When all rows have null values for the rows selected, the aggregation will result in null.

- When you include the **DISTINCT** keyword with the column to be aggregated, duplicate row values for the column are removed, so that only unique values are aggregated.

  distinct_acct_num.sql

  ```
  SELECT COUNT(DISTINCT account_number) AS distinct_accts
    FROM order_header;
  ```

The following query will return the total number of orders ever placed.

order_count.sql
```
SELECT COUNT(*) FROM order_header;
```

This query will return the number of orders that were placed on account, as opposed to orders paid for with cash, check, etc., which have null account numbers.

account_orders.sql
```
SELECT COUNT(account_number) AS order_on_acct
  FROM order_header;
```

This last query will return the number of unique accounts that have been used for purchases.

distinct_acct_num.sql
```
SELECT COUNT(DISTINCT account_number) AS distinct_accts
  FROM order_header;
```

GROUPING ROWS

- Use **GROUP BY** to evaluate an aggregate function over groups of rows.

  group_by.sql
  ```
  SELECT invoice_number, SUM(quantity * price)
    FROM order_item oi JOIN product p USING (product_id)
   GROUP BY invoice_number;
  ```

- The rows are organized into groups in which the values of the grouping column are equal.

  - The aggregate function is then evaluated once for each group.

  - The **SELECT** statement returns a single summary row for each group.

  - Only aggregate functions and the column(s) used in the **GROUP BY** clause are permitted in the **SELECT** clause.

- **HAVING** can be used to eliminate groups based on the value of their aggregations.

  having.sql
  ```
  SELECT invoice_number, SUM (quantity * price)
    FROM order_item oi JOIN product p USING (product_id)
   GROUP BY invoice_number
  HAVING SUM (quantity * price) > 2000;
  ```

group_by2.sql

```
SELECT pay_type_code, AVG(pay_amount) "average pay"
  FROM employee
 GROUP BY pay_type_code;
```

The **GROUP BY** clause comes after any **WHERE** clause:

group_by3.sql

```
SELECT invoice_number, SUM (quantity * price)
  FROM order_item oi JOIN product p USING (product_id)
 WHERE vendor_id = 'DELL'
 GROUP BY invoice_number;
```

Multiple columns in the **GROUP BY** clause will produce subgroups.

group_by4.sql

```
SELECT store_number, title, COUNT(*) "number of employees"
  FROM employee
 GROUP BY store_number, title;
```

group_by5.sql

```
SELECT pay_type_name, AVG(pay_amount) "average pay",
       MAX(pay_amount) "max pay", MIN(pay_amount) "min pay"
  FROM employee JOIN pay_type USING (pay_type_code)
 GROUP BY pay_type_name;
```

COMBINING SELECT STATEMENTS

- Set operators combine the results of two or more queries into one.
  - The **SELECT** lists of all queries must have the same number and data type of expressions in the select list, in the same order.
  - An **ORDER BY** clause can appear after the final query.
- The **UNION** (***OR***) operator combines the results of both queries into a single resultset, but returns only distinct records.

  union.sql

```
SELECT manager_id, firstname, lastname
  FROM person JOIN store ON manager_id = id
 UNION
SELECT vendor_rep_id, firstname, lastname
  FROM person JOIN vendor ON vendor_rep_id = id;
```

  - The **UNION ALL** operator will retain duplicates.
- The **INTERSECT** (***AND***) operator returns rows common to both.

  intersect.sql

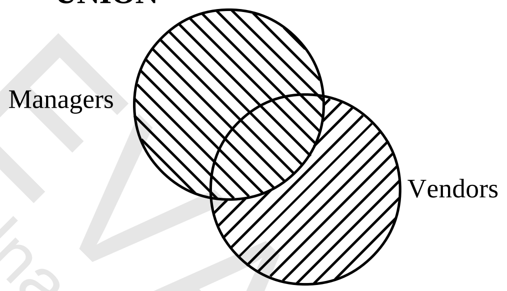
```
SELECT customer_id, firstname, lastname
  FROM person JOIN account ON customer_id = id
INTERSECT
SELECT vendor_rep_id, firstname, lastname
  FROM person JOIN vendor ON vendor_rep_id = id;
```

- The **MINUS** (***AND NOT***) operator returns all rows in the first query that are not in the second; this is yet another way of producing an antijoin.

  minus.sql

```
SELECT id, firstname, lastname
  FROM person
 MINUS
SELECT id, firstname, lastname
  FROM person JOIN employee USING (id);
```

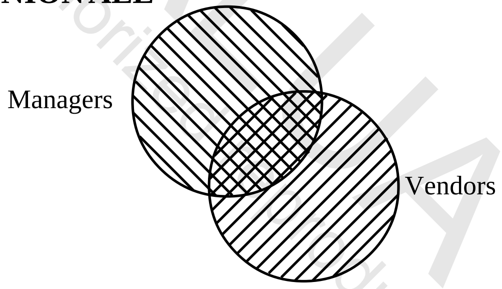
UNION



union2.sql

```
SELECT manager_id
  FROM store
 UNION
SELECT vendor_rep_id
  FROM vendor;
```

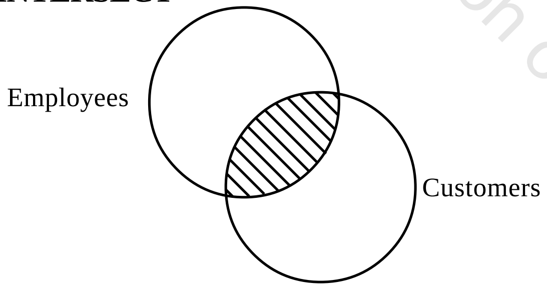
UNION ALL



union_all.sql

```
   SELECT manager_id
     FROM store
UNION ALL
   SELECT vendor_rep_id
     FROM vendor;
```

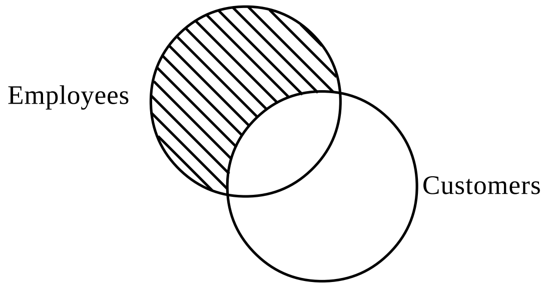
INTERSECT



intersect2.sql

```
   SELECT id
     FROM employee
INTERSECT
   SELECT customer_id
     FROM order_header;
```

MINUS



minus2.sql

```
SELECT id
  FROM employee
 MINUS
SELECT customer_id
  FROM order_header;
```

When you combine two queries with a set operator, Oracle uses the column names (or aliases) from the first query for the final result set.

LABS

❶ Write queries that will:

a. Retrieve the total number of employees at Store #1.
(Solution: *report_a.sql*)

b. Retrieve the total number of supervisor ids at Store #1.
(Solution: *report_b.sql*)

c. Retrieve the total number of store managers.
(Solution: *report_c.sql*)

d. Report the average wage for hourly workers.
(Solution: *report_d.sql*)

e. Report all hourly employees making less than the average hourly wage.
(Solution: *report_e.sql*)

f. Report the names of the highest paid salaried employees of the company.
(Solution: *report_f.sql*)

g. Report the names of the lowest paid salaried employees of the company.
(Solution: *report_g.sql*)

h. Report the total balance owed on all accounts.
(Solution: *report_h.sql*)

i. Report the total amount due on all orders at Store #1.
(Solution: *report_i.sql*)

❷ Write queries that will:

a. Report the average employee income based on state and pay type.
(Solution: *avg_emp_income.sql*)

b. Report the number of employees for each store.
(Solution: *num_emps.sql*)

c. Report the number of employees for each store that has more than 15 employees.
(Solution: *num_emps2.sql*)

❸ Write a query that will retrieve a combined list of the ids and the names of all store managers and vendor sales reps.
(Solution: *mgr_vendors.sql*)