Oracle SQL

with examples and
hands-on exercises

WEBUCATOR

Version: 1.6.2

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Stephen Withrow (Editor)

Stephen has over 30 years of experience in training, development, and consulting in a variety of technology areas including Python, Java, C, C++, XML, JavaScript, Tomcat, JBoss, Oracle, and DB2. His background includes design and implementation of business solutions on client/server, Web, and enterprise platforms. Stephen has a degree in Computer Science and Physics from Florida State University.

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/ORA100-1.6.2.zip.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata/.

Table of Contents

LESSON 1
Relational Database Basics

Topics Covered

☑ History of SQL and relational databases.

☑ Structure of relational databases.

☑ Most popular relational databases.

☑ Oracle schemas and users.

Introduction

SQL stands for Structured Query Language and is pronounced either *ess-que-el* or *sequel*. It is the language used by relational database management systems (RDBMS) to access and manipulate data and to create, structure and destroy databases and database objects.

1.1. Brief History of SQL

In 1970, Dr. E.F. Codd published "A Relational Model of Data for Large Shared Data Banks," an article that outlined a model for storing and manipulating data using tables. Shortly after Codd's article was published, IBM began working on creating a relational database. Between 1979 and 1982, Oracle (then Relational Software, Inc.), Relational Technology, Inc. (later acquired by Computer Associates), and IBM all put out commercial relational databases, and by 1986 they all were using SQL as the data query language.

In 1986, the American National Standards Institute (ANSI) oversaw the standardization of SQL. This standard is updated every few years. Standard SQL is sometimes called ANSI SQL. All major relational databases support this standard but each has its own proprietary extensions. Our focus is on Oracle's flavor of SQL.

1.2. Relational Databases

A relational database at its simplest is a set of tables used for storing data. Each table has a unique name and may relate to one or more other tables in the database through common values.

❖ 1.2.1. Tables

A table in a database is a collection of rows and columns. Tables are also known as entities or relations.

❖ 1.2.2. Rows

A row contains data pertaining to a single item or record in a table. Rows are also known as records, instances, or tuples.

❖ 1.2.3. Columns

A column contains data representing a specific characteristic of the records in the table. Columns are also known as fields, properties, or attributes.

❖ 1.2.4. Relationships

A relationship is a link between two tables (i.e, relations). Relationships make it possible to find data in one table that pertains to a specific record in another table.

❖ 1.2.5. Data Types

Each of a table's columns has a defined data type that specifies the type of data that can exist in that column. For example, the `first_name` column might be defined as `VARCHAR2(20)`, indicating that it can contain a string of up to 20 characters. Unfortunately, data types vary widely between databases.

❖ 1.2.6. Primary Keys

Most tables have a column or group of columns that can be used to uniquely identify records. These columns are called primary keys. For example, an `employees` table might have a column called `employee_id` that is unique for every row. This makes it easy to keep track of a record over time and to associate a record with records in other tables.

❖ 1.2.7. Foreign Keys

Foreign key columns are columns that link to primary key columns in other tables, thereby creating a relationship. For example, the `customers` table might have a foreign key column called `sales_rep` that links to `employee_id`, the primary key in the `employees` table.

❖ 1.2.8. Relational Database Management System

A Relational Database Management System (RDBMS), commonly (but incorrectly) called a database, is software for creating, manipulating, and administering a database. For simplicity, we will often refer to RDBMSs as databases.

1.3. Popular Databases

❖ 1.3.1. Commercial Databases

Oracle

Oracle is the most popular relational database. It runs on both Unix and Windows. It used to be many times more expensive than SQL Server and DB2, but it has come down a lot in price. Oracle now offers a scaled down version that is free for personal or business use: Oracle XE (Extended Edition).

SQL Server

SQL Server is Microsoft's database and, not surprisingly, only runs on Windows. It has only a slightly higher market share than Oracle on Windows machines. Many people find it easier to use than Oracle.

DB2

IBM's DB2 was one of the earliest players in the database market. It is still very commonly used on mainframes and runs on both Windows and Unix.

❖ 1.3.2. Popular Open Source Databases

MySQL

Because of its small size, its speediness, and its very good documentation, MySQL has quickly become the most popular open source database. MySQL is available on both Windows and Unix.

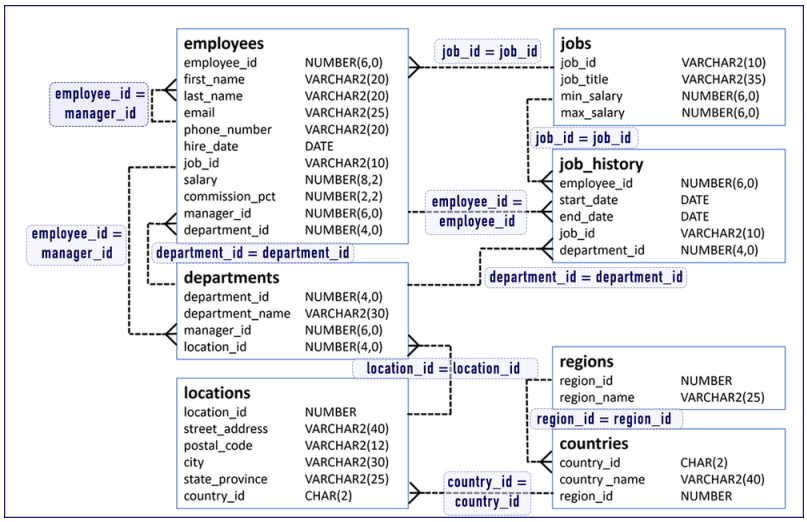
PostgreSQL

PostgreSQL calls itself "the world's most advanced Open Source relational database." It is certainly a featureful and robust database management system and a good choice for people who want some of the advanced features that MySQL doesn't yet have.

1.4. Schemas and Users

A schema is a collection of tables, views, procedures, indexes and other logical objects. Each schema is owned by a database user and has the same name as that user.

Because there is essentially a 1-to-1 relationship between schemas and users, you can think of a schema as a user and a user as a schema. When you did your setup, you should have created the `HR` user and then run a couple of scripts to create and populate schema objects for a fictional Human Resources company. A simple entity relationship diagram showing the relationships between the tables in this schema is shown below:



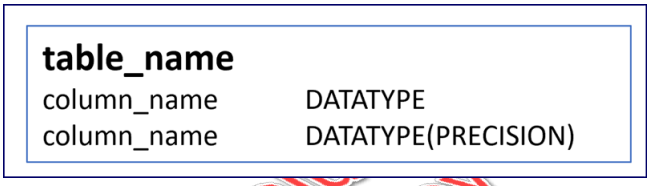
This diagram is included in a PDF called `hr-entity-diagram.pdf` in your class files. We recommend you print that out to have as a reference as you go through the rest of the lessons.

❖ 1.4.1. Connection Lines

Note that the connection lines only show the relationships between the tables. They **do not** point to the specific fields within each table that make the connection. Often you can tell which fields make the connection because they share the same names (e.g., `region_id` in the `regions` and `countries` tables). But this is not always the case. For example, the `manager_id` field in the `departments` table is a foreign key connecting to the `employee_id` field in the `employees` table.

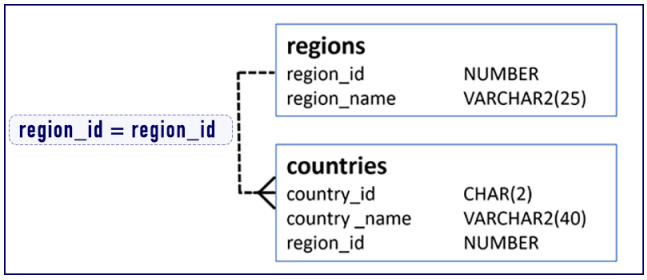
❖ 1.4.2. Tables

Each table in the schema is represented as follows:



A table is broken into columns (or fields), each of which is of a certain data type. In some cases, the precision or size of that data type (e.g., the number of characters) is specified in parentheses.

Take some time to study the relationships in the diagram. Note that the connection lines show one-to-many relationships with the many side ending with a *crow's foot* (a branched line). For example, many countries can be in a single region:



Oracle's HR Schema

Oracle has not updated the data in its HR schema for many years. We expect this is so that all the documentation and samples that use this schema do not become out of sync with the data.

This is a good thing; however, sometimes it makes the data feel old, which it is. Not to worry. All the concepts taught in these lessons are up to date as of version Oracle 19c.

Conclusion

We have covered a little bit of the history of SQL, how databases work, and the common SQL statements. Now we will get into learning how to work with SQL.

LESSON 2
Creating Tables

Topics Covered

☑ Standard data types.

☑ `CREATE TABLE`.

☑ `ALTER TABLE`.

☑ `DROP TABLE`.

Introduction

At its most basic, creating tables involves specifying the name of the table, the fields the table contains and the data types of those fields. First, we'll take a look at data types and then we'll look at the syntax for creating, modifying and deleting tables.

2.1. Data Types

Oracle allows for many different data types. The most common ones are shown below:

Common Oracle Data Types

Data Type	Description
`CHAR [(size)]`	Character data with length of `size`, which defaults to 1.
`NCHAR [(size)]`	Unicode character data with length of `size`, which defaults to 1.
`VARCHAR2(size)`	Character data with length of up to `size`.
`NVARCHAR2(size)`	Unicode character data with length of up to `size`.
`LONG`	Character data of variable length up to 2 gigabytes. Used to store large amounts of data.
`NUMBER [(p[,s])]`	A number. The precision (`p`) specifies the total number of digits allowed. The scale (`s`) specifies the number of digits after the decimal point.
`FLOAT [(p)]`	A number with a floating point (i.e., a decimal). The precision (`p`) specifies the total number of digits allowed.
`DATE`	A date.
`TIMESTAMP`	A specific point in time.

Square Brackets in Code Notation

Square brackets in code notation means that the contained portion is optional. To illustrate, consider the following from the table above:

```
NUMBER [(p[,s])]
```

This means that all of the following are valid ways of expressing a number:

1. `NUMBER` - No precision is specified.
2. `NUMBER(6)` - Precision is specified, but scale is not.
3. `NUMBER(6,3)` - Precision and scale are specified.

Note that scale cannot be specified unless precision is also specified. The outside square brackets in `[(p[,s])]` indicate that the whole section is optional. The inside square brackets indicate that scale is optional even if precision is specified. If it were written as `[(p,s)]`, it would indicate that precision and scale are optional, but if one is included, the other must be included as well.

See `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Data-Types.html` for complete information on Oracle SQL data types.

2.2. Creating Tables

Now that you understand data types, you're ready to learn how to create a table. Note that you must have the proper database permissions to do so.

```
CREATE TABLE table_name
(
  column_name DATATYPE,
  column_name DATATYPE,
  column_name DATATYPE,
  more columns...
);
```

❖ 2.2.1. NULL Values

As we have discussed, in some cases a value may not be set for a specific field in a specific record. In this case, that field contains `NULL`. In the `CREATE TABLE` statement, the default for each column is to allow for `NULL` values. This can, but doesn't have to be, explicitly stated with the `NULL` flag or it can be changed with the `NOT NULL` flag.

Forcing Values with 'NOT NULL'

```
CREATE TABLE table_name
(
  column_name DATATYPE NOT NULL,
  column_name DATATYPE NULL,
  column_name DATATYPE NOT NULL,
  more columns...
);
```

❖ 2.2.2. Primary Keys

Every table should have a primary key, which is a field or group of fields that can be used to uniquely identify specific records in the table. Primary keys can be specified in two ways as shown below:

Specifying a Primary Key: Method 1

```
CREATE TABLE table_name
(
  column_name DATATYPE NOT NULL PRIMARY KEY,
  column_name DATATYPE,
  column_name DATATYPE NOT NULL,
  more columns...
);
```

Specifying a Primary Key: Method 2

```
CREATE TABLE table_name
(
  column_name_pk DATATYPE NOT NULL,
  column_name DATATYPE,
  column_name DATATYPE NOT NULL,
  more columns...,
  CONSTRAINT PK_table_name PRIMARY KEY (column_name_pk)
);
```


When a table uses a group of fields for the primary key, the key is called a *composite primary key*. In this case, you must specify the primary key at the end of the `CREATE TABLE` statement as shown below:

```
CREATE TABLE table_name
(
  column_name_pk1 data_type NOT NULL,
  column_name_pk2 data_type NOT NULL,
  column_name data_type,
  more columns...,
  CONSTRAINT PK_table_name PRIMARY KEY (column_name_pk1, column_name_pk2)
);
```

IDENTITY Clause

As of Oracle 12c, it is possible to create an auto-incrementing field when creating a table. This makes it easy to generate unique primary keys. This is done using `NUMBER GENERATED AS IDENTITY` as follows:

```
CREATE TABLE table_name
(
  column_name_pk NUMBER GENERATED AS IDENTITY,
  column_name DATATYPE,
  column_name DATATYPE NOT NULL,
  more columns...,
  CONSTRAINT PK_table_name PRIMARY KEY (column_name_pk)
);
```

See `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html` for additional options for the `IDENTITY` clause.

❖ 2.2.3. Foreign Keys

Foreign keys reference the primary key of another table, thereby creating a relationship between those two tables. They can be specified using the two methods shown below:

Specifying a Foreign Key: Method 1

```
CREATE TABLE table_name
(
  column_name data_type NOT NULL PRIMARY KEY,
  column_name data_type,
  column_name_fk data_type REFERENCES table2_name(column_name_pk),
  more columns...
);
```


Specifying a Foreign Key: Method 2

```
CREATE TABLE table_name
(
  column_name data_type NOT NULL PRIMARY KEY,
  column_name data_type,
  column_name_fk data_type,
  more columns...,
  CONSTRAINT FK_table_name REFERENCES table2_name(column_name_pk)
);
```

The code samples below show how to create duplicates of the `regions` and `countries` tables, which we have named `regions_copy` and `countries_copy`, respectively.

Demo 2.1: Creating-Tables/Demos/create-table.sql

```
1.   CREATE TABLE regions_copy
2.   (
3.       region_id   NUMBER      NOT NULL    PRIMARY KEY,
4.       region_name VARCHAR2(25)
5.   );
6.
7.   CREATE TABLE countries_copy
8.   (
9.       country_id      CHAR(2) NOT NULL    PRIMARY KEY,
10.      country_name    VARCHAR2(40),
11.      region_id       NUMBER  REFERENCES regions_copy(region_id)
12.  );
```

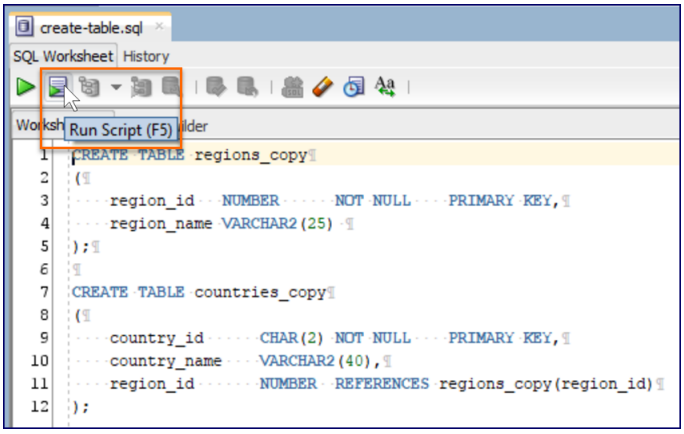
Code Explanation

Notice that both tables have primary keys and that the `countries_copy` table has a foreign key referencing the `regions_copy` table.

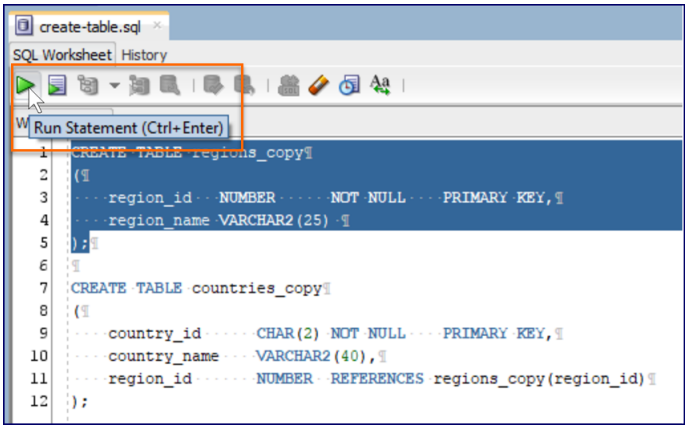
Run the code in SQL Developer to create the tables.

Running Code in SQL Developer

When you have a SQL file open in SQL Developer, you can run the whole file by pressing **F5** or clicking the **Run Script** icon:



To run a portion of the file, highlight the portion you want to run and click **Ctrl+Enter** or click the **Run Statement** icon:



If you like, you can populate these tables with the same data from the corresponding `regions` and `countries` tables by running `Demos/populate-tables.sql`. Don't worry about how that code works yet; we'll cover it in detail later.

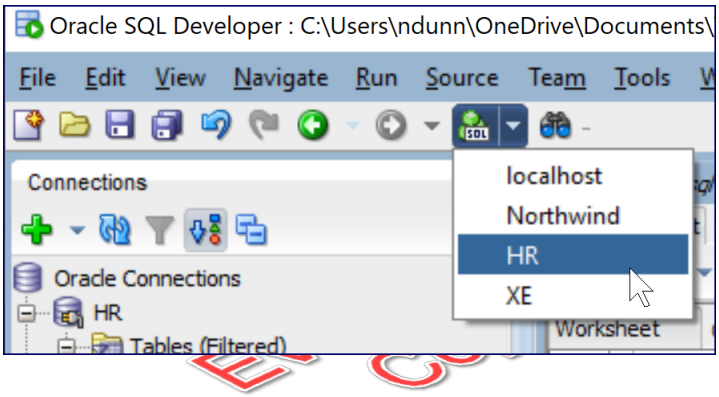
Exercise 1: Creating Tables

20 to 30 minutes

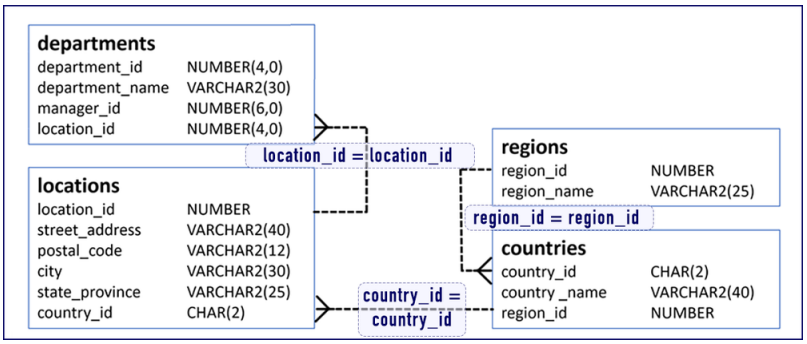
In this exercise you will create duplicates of the `locations` and `departments` tables, which you should name `locations_copy` and `departments_copy`, respectively.

Opening a New SQL Worksheet in SQL Developer

To open a new SQL worksheet in SQL Developer click the down arrow on the right of the **SQL icon** and select **HR**:



1. If you have not yet executed `Creating-Tables/Demos/create-table.sql`, do so now to create the `countries_copy` and `regions_copy` tables.
2. Open a new SQL Worksheet in SQL Developer and save it as `create-tables.sql` in `Creating-Tables/Exercises`.
3. Write the two `CREATE TABLE` statements using the entity diagram below as a guide, but append "_copy" to the end of the table names:



4. Run the file to see if it executes. If you get any errors, fix them and try again.

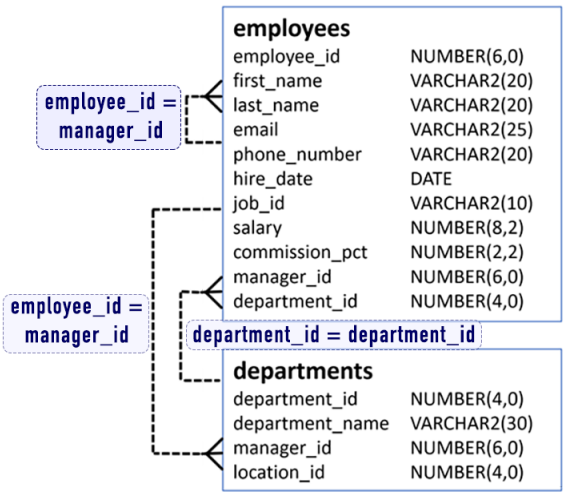
5. Run the code in `Exercises/populate-tables.sql` to populate your tables. If this code works without erroring then you likely created your tables correctly.

Solution: Creating-Tables/Solutions/create-table.sql

```
1.   CREATE TABLE locations_copy
2.   (
3.     location_id     NUMBER      NOT NULL    PRIMARY KEY,
4.     street_address  VARCHAR2(40),
5.     postal_code     VARCHAR2(40),
6.     city            VARCHAR2(40),
7.     state_province  VARCHAR2(40),
8.     country_id      CHAR(2) REFERENCES countries_copy(country_id)
9.   );
10.
11.  CREATE TABLE departments_copy
12.  (
13.    department_id   NUMBER(4,0) NOT NULL    PRIMARY KEY,
14.    department_name VARCHAR2(30),
15.    manager_id     NUMBER(6,0),
16.    location_id    NUMBER(4,0) REFERENCES locations_copy(location_id)
17.  );
```


2.3. Adding Constraints

Examine the relationships between the `departments` and `employees` tables:



Notice that the `manager_id` in the `employees` table references the `employee_id` in the same table. We refer to this type of reference as a "self-referencing join". In this relationship, the `manager_id` field

is a foreign key. Each employee can only have a single manager (or no manager at all), and can be the manager of zero or more other employees. Because this relationship cannot be made until the table exists, we have to first create the `employees` table and then add the relationship after using `ALTER TABLE`:

We can use the code below to create the table:

Demo 2.2: Creating-Tables/Demos/create-employees.sql

```
1.   CREATE TABLE employees_copy
2.   (
3.     employee_id     NUMBER(6)       NOT NULL    PRIMARY KEY,
4.     first_name      VARCHAR2(20),
5.     last_name       VARCHAR2(25)    NOT NULL,
6.     email           VARCHAR2(25)    NOT NULL,
7.     phone_number    VARCHAR2(20),
8.     hire_date       DATE            NOT NULL,
9.     job_id          VARCHAR2(10)    NOT NULL,
10.    salary          NUMBER(8,2),
11.    commission_pct  NUMBER(2,2),
12.    manager_id      NUMBER(6),
13.    department_id  NUMBER(4) REFERENCES departments_copy(department_id)
14.  );
```


Code Explanation

Notice that the `manager_id` field is *not* a foreign key.

We then can add the foreign key constraint using `ALTER TABLE`:

Demo 2.3: Creating-Tables/Demos/alter-employees.sql

```
1.   ALTER TABLE employees_copy
2.   ADD CONSTRAINT employee_manager_fk
3.   FOREIGN KEY (manager_id)
4.   REFERENCES employees_copy(employee_id);
```

Code Explanation

When creating a constraint in this way, you must name the constraint. We named ours `employee_manager_fk`.

❖ 2.3.1. Dropping Constraints

Dropping constraints is simple:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

If you don't know the name of the constraint that you want to remove, you can find all constraints on a table using the following query:

```
SELECT uc.constraint_name, uc.constraint_type, uc.search_condition
FROM user_cons_columns ucc
JOIN user_constraints uc ON uc.constraint_name = ucc.constraint_name
WHERE upper(ucc.table_name) = 'YOUR_TABLE_NAME';
```

For example, to find all the constraints on the `employees_copy` table that we created earlier, we would use this query:

Demo 2.4: Creating-Tables/Demos/find-constraints.sql

```
1.    SELECT uc.constraint_name, uc.constraint_type, uc.search_condition
2.    FROM user_cons_columns ucc
3.    JOIN user_constraints uc ON uc.constraint_name = ucc.constraint_name
4.    WHERE upper(ucc.table_name) = 'EMPLOYEES_COPY';
```

Code Explanation

This would return the following:

	CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION
1	SYS_C008400	C	"EMPLOYEE_ID" IS NOT NULL
2	SYS_C008401	C	"LAST_NAME" IS NOT NULL
3	SYS_C008402	C	"EMAIL" IS NOT NULL
4	SYS_C008403	C	"HIRE_DATE" IS NOT NULL
5	SYS_C008404	C	"JOB_ID" IS NOT NULL
6	SYS_C008405	P	(null)

Don't Worry about the SQL

Do not worry about the SQL in the `SELECT` queries yet. You will understand it later. The important takeaway is that you can use this query to get the names of your constraints.

Exercise 2: Altering the departments_copy Table

5 to 10 minutes

Notice that the `departments_copy` table has a `manager_id` field. As we created this table before creating the `employees_copy` table, we couldn't make the `manager_id` field a foreign key referencing the `employees_copy` table.

1. Open a new SQL Worksheet in SQL Developer and save it as `alter-table.sql` in `Creating-Tables/Exercises`.
2. Write a statement to add a foreign key constraint to the `manager_id` field in the `departments_copy` table.
3. Run the file to see if it executes. If you get any errors, fix them and try again.

Solution: Creating-Tables/Solutions/alter-table.sql

```
1.    ALTER TABLE departments_copy
2.    ADD CONSTRAINT department_manager_fk
3.    FOREIGN KEY (manager_id)
4.    REFERENCES employees_copy(employee_id);
```

2.4. UNIQUE Constraints

A `UNIQUE` constraint forces all values in a field (column) to be unique. For example, the following code would add a `UNIQUE` constraint to the `email` field in the `employees` table, which would prevent duplicate email addresses:

```
ALTER TABLE employees_copy
ADD CONSTRAINT emp_email_uk UNIQUE (email);
```


Note that you can also add the `UNIQUE` constraint when originally creating a table in either of the following ways:

```
CREATE TABLE employees_copy
(
  ...,
  email VARCHAR2(25) NOT NULL UNIQUE,
  ...
);
```

```
CREATE TABLE employees
(
  ...,
  email VARCHAR2(25) NOT NULL,
  ...
  CONSTRAINT emp_email_uk UNIQUE (email)
  ...
);
```

Custom Constraints

In addition to `NOT NULL`, `UNIQUE`, and foreign key constraints, you can create custom (`CHECK`) constraints forcing field values to meet specific conditions. For example, you could set minimum and maximum values for a `NUMBER` field.

2.5. Adding and Dropping Columns

Adding and dropping columns is similar to adding and dropping constraints. To add a column, use `ALTER TABLE` as shown below:

```
ALTER TABLE table_name
ADD column_name column_definition;
```

And to drop the column:


```
ALTER TABLE table_name
DROP COLUMN column_name;
```

For example, the code shown below first adds a `birth_date` column to `employees_copy` and then drops it:

Demo 2.5: Creating-Tables/Demos/add-drop-column.sql

```
1.   ALTER TABLE employees_copy
2.   ADD birth_date DATE;
3.
4.   ALTER TABLE employees_copy
5.   DROP COLUMN birth_date;
```

2.6. Dropping Tables

Dropping tables is scarily easy:

```
DROP TABLE table_name
```

Luckily, there are data integrity checks to make sure you don't drop tables that are referenced by other tables. For example, if you try to drop `employees_copy`, you will get an error like the one below:

```
Error starting at line : 1 in command -
DROP TABLE employees_copy
Error report -
ORA-02449: unique/primary keys in table referenced by foreign keys
02449. 00000 -  "unique/primary keys in table referenced by foreign keys"
*Cause:    An attempt was made to drop a table with unique or
           primary keys referenced by foreign keys in another table.
*Action:   Before performing the above operations the table, drop the
           foreign key constraints in other tables. You can see what
           constraints are referencing a table by issuing the following
           command:
           SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";
```

So, to drop a table, you must first drop the foreign key constraints referencing the table. The query recommended in the error shown above can be improved to just get the foreign-key constraints by using the "R" constraint-type key:

Demo 2.6: Creating-Tables/Demos/find-fk-constraints.sql

```
1.   SELECT uc.constraint_name, uc.constraint_type, ucc.column_name
2.   FROM user_cons_columns ucc
3.   JOIN user_constraints uc ON uc.constraint_name = ucc.constraint_name
4.   WHERE upper(ucc.table_name) IN ('COUNTRIES_COPY',
5.                         'DEPARTMENTS_COPY',
6.                         'EMPLOYEES_COPY',
7.                         'LOCATIONS_COPY',
8.                         'REGIONS_COPY')
9.       AND uc.constraint_type = 'R'
10.  ORDER BY ucc.table_name;
```

Code Explanation

Do not worry about how this `SELECT` statement works. We will cover `SELECT` statements in detail throughout these lessons.

Run that query and you should get results similar to the ones shown below:

	CONSTRAINT_NAME	CONSTRAINT_TYPE	COLUMN_NAME
1	SYS_C008412	R	REGION_ID
2	SYS_C008418	R	LOCATION_ID
3	DEPARTMENT_MANAGER_FK	R	MANAGER_ID
4	EMPLOYEE_MANAGER_FK	R	MANAGER_ID
5	SYS_C008325	R	DEPARTMENT_ID
6	SYS_C008415	R	COUNTRY_ID

You can now run the `ALTER TABLE` statements in `Creating-Tables/Demos/drop-tables.sql` to drop the constraints. The queries will be similar to these:

Demo 2.7: Creating-Tables/Demos/drop-tables.sql

```
1.   -- BEFORE RUNNING THE ALTER TABLE STATEMENTS BELOW,
2.   -- YOU WILL NEED TO CHANGE THE CONSTRAINT NAMES TO MATCH THE ONES
3.   -- RETURNED BY THE QUERY IN find-fk-constraints.sql
4.   ALTER TABLE COUNTRIES_COPY DROP CONSTRAINT SYS_C008412;
5.   ALTER TABLE DEPARTMENTS_COPY DROP CONSTRAINT DEPARTMENT_MANAGER_FK;
6.   ALTER TABLE DEPARTMENTS_COPY DROP CONSTRAINT SYS_C008418;
7.   ALTER TABLE EMPLOYEES_COPY DROP CONSTRAINT EMPLOYEE_MANAGER_FK;
8.   ALTER TABLE EMPLOYEES_COPY DROP CONSTRAINT SYS_C008325;
9.   ALTER TABLE LOCATIONS_COPY DROP CONSTRAINT SYS_C008415;
10.
11.
12.  -- FINALLY, YOU CAN DROP THE TABLES
13.  DROP TABLE regions_copy;
14.  DROP TABLE countries_copy;
15.  DROP TABLE locations_copy;
16.  DROP TABLE departments_copy;
17.  DROP TABLE employees_copy;
```


Code Explanation

Note that you must update the queries to use the constraint names returned by the `SELECT` query above.

And finally, after running the `ALTER TABLE` statements to drop the constraints, you can run the `DROP TABLE` statements to drop the tables.

Conclusion

In this lesson, you have learned how to create tables, how to set data types and how to create relationships between tables using primary and foreign keys. You have also learned how to add and remove constraints and columns.

LESSON 3
Basic Selects

Topics Covered

☑ Comments.

☑ Selecting all rows.

☑ Sorting.

☑ Filtering.

Introduction

The `SELECT` statement is used to retrieve data from tables. `SELECT` statements can be used to perform simple tasks such as retrieving records from a single table or complicated tasks such as retrieving data from multiple tables with record grouping and sorting. In this lesson, we will look at several of the more basic ways to retrieve data from a single table.

3.1. Comments

Oracle allows for two types of comments:

1. Single-line comments begin with two dashes (`--`).
2. Multi-line comments begin with `/*` and end with `*/`.

```
-- This is a single-line comment.

/*
  This is
  a multi-line
  comment.
*/
```

3.2. Whitespace and Semi-colons

Whitespace is ignored in SQL statements. Multiple statements are separated with semi-colons. The two statements in the sample below are equally valid.

Demo 3.1: Basic-Selects/Demos/whitespace.sql

```
1.    SELECT * FROM employees;
2.
3.    SELECT *
4.    FROM employees;
```

3.3. Case Sensitivity

In Oracle, some things are case sensitive and others are not:

1. Keywords (e.g., `SELECT`, `FROM`, `WHERE`, `CREATE TABLE`) **are not** case sensitive. Our convention is to use all uppercase letters for these words.
2. Identifiers (e.g., table, column names, constraint names) **are not** case sensitive. All of the following are valid:


```
SELECT first_name, last_name FROM employees;

SELECT First_Name, Last_Name FROM Employees;

SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEES;
```

Note that when tables are created, Oracle automatically makes identifiers uppercase at the point of creation. So, when you run a `CREATE TABLE` statement to create a table named "employees" with columns named "first_name" and "last_name", Oracle actually creates a table named "EMPLOYEES" with columns named "FIRST_NAME" and "LAST_NAME". However, we follow Oracle's own documentation practice of taking advantage of the fact that Oracle is case insensitive and writing identifiers with all lowercase letters. As such, we prefer the first of the three `SELECT` statements shown above:

```
SELECT first_name, last_name FROM employees;
```

We prefer this convention because the lowercase identifiers stand out from the uppercase keywords.

3. String (i.e., text) comparisons **are** case sensitive. So, if you search for an employee with the first name of "nat" and the record is stored in the table as "Nat", you will not find the result.

3.4. SELECTing All Columns in All Rows

The following syntax is used to retrieve all columns in all rows of a table.

```
SELECT *
FROM table_name;
```

Demo 3.2: Basic-Selects/Demos/select-all.sql

```
1.    -- Retrieve all columns in the regions table
2.    SELECT *
3.    FROM regions;
```


Code Explanation

The above `SELECT` statement will return the following results:

	REGION_ID	REGION_NAME
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa
5	7	Africa

As you can see, the `regions` table has only two columns, `region_id` and `region_name`, and five rows.

Exercise 3: Exploring the Tables

10 to 20 minutes

In this exercise, you will explore all the data in the HR database by selecting all the rows of each of the tables.

1. Open a new SQL Worksheet in SQL Developer and save it as `select-all.sql` in `Basic-Selects/Exercises`.
2. Write the necessary SQL queries to select all columns of all rows from the tables below.
3. The number of records that should be returned is indicated in parentheses next to the table name:
    A. regions (4)
    B. countries (25)
    C. locations (23)
    D. departments (27)
    E. employees (107) - Only the first 50 will load at first. You will need to scroll through the results to see them all.
    F. jobs (19)
    G. job_history (10)

Solution: Basic-Selects/Solutions/select-all.sql

```
1.    SELECT * FROM regions;
2.    SELECT * FROM countries;
3.    SELECT * FROM locations;
4.    SELECT * FROM departments;
5.    SELECT * FROM employees;
6.    SELECT * FROM jobs;
7.    SELECT * FROM job_history;
```

3.5. SELECTing Specific Columns

While using `SELECT *` seems like a handy way to display all of the columns in a table, its use is considered a poor practice in a production environment because of potential performance problems. The following syntax is used to retrieve specific columns in all rows of a table.


```
SELECT column_name, column_name
FROM table_name;
```

Demo 3.3: Basic-Selects/Demos/select-cols.sql

```
1.    /*
2.      Select the first_name and last_name columns
3.      from the employees table.
4.    */
5.    SELECT first_name, last_name
6.    FROM employees;
```

Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME
1	Steven	King
2	Neena	Kochhar
3	Lex	De Haan
4	Alexander	Hunold
5	Bruce	Ernst
6	David	Austin
7	Valli	Pataballa
8	Diana	Lorentz
9	Nancy	Greenberg
10	Daniel	Faviet

Exercise 4: SELECTing Specific Columns

5 to 15 minutes

In this exercise, you will practice selecting specific columns from tables in the HR database.

1. Open a new SQL Worksheet in SQL Developer and save it as `select-cols.sql` in `Basic-Selects/Exercises`.
2. Select `country_name` from the `countries` table.
3. Select `department_id`, `department_name`, `manager_id` from the `departments` table.
4. Select `employee_id`, `start_date`, `end_date`, and `job_id` from the `job_history` table.
5. Select `job_title`, `min_salary`, and `max_salary` from the `jobs` table.
6. Select `street_address`, `city`, `state_province`, and `postal_code` from the `locations` table.
7. Select `region_id` and `region_name` from the `regions` table.

Solution: Basic-Selects/Solutions/select-cols.sql

```
1.   SELECT country_name
2.   FROM countries;
3.
4.   SELECT department_id, department_name, manager_id
5.   FROM departments;
6.
7.   SELECT employee_id, start_date, end_date, job_id
8.   FROM job_history;
9.
10.  SELECT job_title, min_salary, max_salary
11.  FROM jobs;
12.
13.  SELECT street_address, city, state_province, postal_code
14.  FROM locations;
15.
16.  SELECT region_id, region_name
17.  FROM regions;
```


3.6. Sorting Records

The `ORDER BY` clause of the `SELECT` statement is used to sort records.

❖ 3.6.1. Sorting by a Single Column

To sort by a single column, simply name that column in the `ORDER BY` clause.

```
SELECT column_name, column_name
FROM table_name
ORDER BY column_name;
```

Note that columns in the `ORDER BY` clause do not have to appear in the `SELECT` clause.

Demo 3.4: Basic-Selects/Demos/order-by-1.sql

```
1.   /*
2.       Select the first_name and last_name columns from the
3.       employees table. Sort by first_name.
4.   */
5.
6.   SELECT first_name, last_name
7.   FROM employees
8.   ORDER BY last_name;
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME
1	Ellen	Abel
2	Sundar	Ande
3	Mozhe	Atkinson
4	David	Austin
5	Hermann	Baer
6	Shelli	Baida
7	Amit	Banda
8	Elizabeth	Bates
9	Sarah	Bell
10	David	Bernstein

❖ 3.6.2. Sorting By Multiple Columns

To sort by multiple columns, comma-delimit the column names in the `ORDER BY` clause.

```
SELECT column_name, column_name
FROM table_name
ORDER BY column_name, column_name;
```

Demo 3.5: Basic-Selects/Demos/order-by-2.sql

```
1.   /*
2.     Select the first_name, last_name, and email columns from the
3.     employees table. Sort first by last_name and then by first_name.
4.   */
5.
6.   SELECT first_name, last_name, email
7.   FROM employees
8.   ORDER BY last_name, first_name;
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME	EMAIL
1	Ellen	Abel	EABEL
2	Sundar	Ande	SANDE
3	Mozhe	Atkinson	MATKINSO
4	David	Austin	DAUSTIN
5	Hermann	Baer	HBAER
6	Shelli	Baida	SBAIDA
7	Amit	Banda	ABANDA
8	Elizabeth	Bates	EBATES
9	Sarah	Bell	SBELL
10	David	Bernstein	DBERNSTE

❖ 3.6.3. Ascending and Descending Sorts

By default, when an `ORDER BY` clause is used, records are sorted in ascending order. This can be explicitly specified with the `ASC` keyword. To sort records in descending order, use the `DESC` keyword.

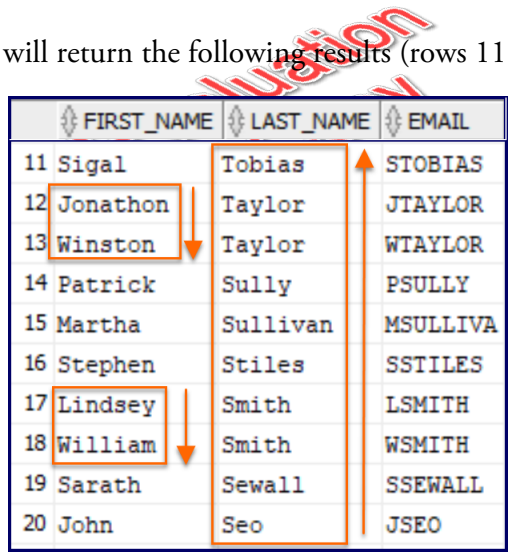
```
SELECT column_name, column_name
FROM table_name
ORDER BY column_name DESC, column_name ASC;
```

Demo 3.6: Basic-Selects/Demos/order-by-3.sql

```
1.   /*
2.     Select the first_name, last_name, and email columns from the
3.       employees table.
4.     Sort first by last_name in desceding order
5.       and then by first_name in ascending order.
6.   */
7.
8.   SELECT first_name, last_name, email
9.   FROM employees
10.  ORDER BY last_name DESC, first_name ASC;
```

Code Explanation

The above `SELECT` statement will return the following results (rows 11 through 20 shown):



	FIRST_NAME	LAST_NAME	EMAIL
11	Sigal	Tobias	STOBIAS
12	Jonathon	Taylor	JTAYLOR
13	Winston	Taylor	WTAYLOR
14	Patrick	Sully	PSULLY
15	Martha	Sullivan	MSULLIVA
16	Stephen	Stiles	SSTILES
17	Lindsey	Smith	LSMITH
18	William	Smith	WSMITH
19	Sarath	Sewall	SSEWALL
20	John	Seo	JSEO

Note that we could, and often would, leave off the `ASC` as that is the default.

Exercise 5: Sorting Results

10 to 20 minutes

In this exercise, you will practice sorting results in `SELECT` statements.

1. Open a new SQL Worksheet in SQL Developer and save it as `sorting.sql` in `Basic-Selects/Exercises`.
2. Select `country_name` from the `countries` table sorted by `region_id`.
3. Select `department_id`, `department_name`, `manager_id` from the `departments` table sorted by `department_name`.
4. Select `employee_id`, `start_date`, `end_date`, and `job_id` from `job_history` and sort by `start_date` (starting with the most recent) and then `end_date` (starting with the most recent) .
5. Create a report showing the minimum and maximum salaries for each job title. Sort by minimum salary (from high to low) and then by maximum salary (from high to low).
6. Create a report showing the street address, city, state or province, and postal code for each location, sorted by state/province and then by city.
7. Create a report showing the region id and name for each region sorted by region name.

Solution: Basic-Selects/Solutions/sorting.sql

```
1.   SELECT country_name
2.   FROM countries
3.   ORDER BY region_id;
4.
5.   SELECT department_id, department_name, manager_id
6.   FROM departments
7.   ORDER BY department_name;
8.
9.   SELECT employee_id, start_date, end_date, job_id
10.  FROM job_history
11.  ORDER BY start_date DESC, end_date DESC;
12.
13.  SELECT job_title, min_salary, max_salary
14.  FROM jobs
15.  ORDER BY min_salary DESC, max_salary DESC;
16.
17.  SELECT street_address, city, state_province, postal_code
18.  FROM locations
19.  ORDER BY state_province, city;
20.
21.  SELECT region_id, region_name
22.  FROM regions
23.  ORDER BY region_name;
```


3.7. The WHERE Clause and Logical Operator Symbols

The `WHERE` clause is used to filter rows; that is, to retrieve specific rows from tables. The `WHERE` clause can contain one or more conditions that specify which rows should be returned.

```
SELECT column_name, column_name
FROM table_name
WHERE conditions;
```

The following table shows the symbolic operators used in `WHERE` conditions.

SQL Symbol Operators

Operator	Description
=	Equals
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

Note that non-numeric values (e.g, dates and strings) in the `WHERE` clause must be enclosed in single quotes and that Oracle treats all character data as case sensitive. Examples are shown below.

❖ 3.7.1. Checking for Equality

Demo 3.7: Basic-Selects/Demos/where-equal.sql

```
1.   /*
2.     Create a report showing the first and last name and salary
3.     of all sales representatives (job_id is 'SA_REP')
4.   */
5.
6.   SELECT first_name, last_name, salary
7.   FROM employees
8.   WHERE job_id = 'SA_REP';
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME	SALARY
1	Peter	Tucker	10000
2	David	Bernstein	9500
3	Peter	Hall	9000
4	Christopher	Olsen	8000
5	Nanette	Cambrault	7500
6	Oliver	Tuvault	7000
7	Janette	King	10000
8	Patrick	Sully	9500
9	Allan	McEwen	9000
10	Lindsey	Smith	8000

❖ 3.7.2. Checking for Inequality

Demo 3.8: Basic-Selects/Demos/where-not-equal.sql

```
1.   /*
2.     Create a report showing the first and last name,
3.     job id, and salary of all employees excluding
4.     sales representatives (job_id is 'SA_REP')
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE job_id <> 'SA_REP';
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	Steven	King	AD_PRES	24000
2	Neena	Kochhar	AD_VP	17000
3	Lex	De Haan	AD_VP	17000
4	Alexander	Hunold	IT_PROG	9000
5	Bruce	Ernst	IT_PROG	6000
6	David	Austin	IT_PROG	4800
7	Valli	Pataballa	IT_PROG	4800
8	Diana	Lorentz	IT_PROG	4200
9	Nancy	Greenberg	FI_MGR	12008
10	Daniel	Faviet	FI_ACCOUNT	9000

Escaping Apostrophes

Imagine that you have an employee whose last name is O'Reilly. To search for this person in the `employees` table, you might try the following `WHERE` clause:

```
WHERE last_name = 'O'Reilly'
```

But that will cause an error, because the apostrophe following the "O" works as a closing single quotation. To correct this, you must escape the apostrophe by putting another apostrophe in front of it:

```
WHERE last_name = 'O''Reilly'
```

Exercise 6: Using the WHERE Clause to Check for Equality or Inequality

10 to 20 minutes

In this exercise, you will practice using the `WHERE` clause to check for equality and inequality.

1. Open a new SQL Worksheet in SQL Developer and save it as `equality-and-inequality.sql` in `Basic-Selects/Exercises`.
2. Create a report showing all the minimum and maximum salaries for sales representatives.
3. Create a report showing the city and province of locations in the United Kingdom (country id: `UK`).
4. Using the `job_history` table, create a report showing the employee id and start and end dates of jobs that ended on December 31, 2006 (`31-DEC-06`).
5. Create a report showing the department name and location id for all departments not in location id 1700.

Solution: Basic-Selects/Solutions/equality-and-inequality.sql

```
1.   SELECT min_salary, max_salary
2.   FROM jobs
3.   WHERE job_title = 'Sales Representative';
4.
5.   SELECT city, state_province
6.   FROM locations
7.   WHERE country_id = 'UK';
8.
9.   SELECT employee_id, start_date, end_date
10.  FROM job_history
11.  WHERE end_date = '31-DEC-06'
12.
13.  SELECT department_name, location_id
14.  FROM departments
15.  WHERE location_id <> 1700;
```

3.8. Checking for Greater Than or Less Than

The less than (<) and greater than (>) signs are used to compare numbers, dates, and strings.

Combine these with the equals operator for less than or equal to (<=) and greater than or equal to (>=).

Demo 3.9: Basic-Selects/Demos/where-greater-than-or-equal.sql

```
1.   /*
2.       Create a report showing the first and last name of all
3.       employees whose last names start with a letter in the
4.       last half of the alphabet.
5.   */
6.
7.   SELECT first_name, last_name
8.   FROM employees
9.   WHERE last_name >= 'N';
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME
1	Valli	Pataballa
2	Ismael	Sciarra
3	Jose Manuel	Urman
4	Luis	Popp
5	Den	Raphaely
6	Sigal	Tobias
7	Matthew	Weiss
8	Shanta	Vollman
9	Julia	Nayer
10	TJ	Olson

Exercise 7: Using the WHERE Clause to Check for Greater or Less Than

5 to 15 minutes

In this exercise, you will practice using the `WHERE` clause to check for values greater than or less than a specified value.

1. Open a new SQL Worksheet in SQL Developer and save it as `greater-than-less-than.sql` in `Basic-Selects/Exercises`.
2. Create a report showing the first and last names and the hire date of all employees hired before January 1, 2003 (`01-JAN-03`).
3. Create a report that shows the job title and the minimum and maximum salaries of all jobs with a maximum salary of at least 10000.
4. Create a report that demonstrates that there are no records in the jobs table that have minimum salaries that are higher than the maximum salaries.

Solution: Basic-Selects/Solutions/greater-than-less-than.sql

```
1.   SELECT first_name, last_name, hire_date
2.   FROM employees
3.   WHERE hire_date < '01-JAN-03';
4.
5.   SELECT job_title, min_salary, max_salary
6.   FROM jobs
7.   WHERE max_salary >= 10000;
8.
9.   SELECT *
10.  FROM jobs
11.  WHERE min_salary > max_salary;
```

3.9. Checking for Null and Not Null

When a field in a row has no value, it is said to be `NULL`. This is not the same as having an empty string or a 0 Rather, it means that the field contains no value at all. When checking to see if a field is `NULL`, you cannot use the equals sign (=); rather, use the `IS NULL` expression.

Demo 3.10: Basic-Selects/Demos/where-null.sql

```
1.   /*
2.     Create a report showing the first and last names and
3.     department id of all employees whose department is unspecified.
4.   */
5.
6.   SELECT first_name, last_name, department_id
7.   FROM employees
8.   WHERE department_id IS NULL;
```

Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
1	Kimberely	Grant	(null)

Demo 3.11: Basic-Selects/Demos/where-not-null.sql

```
1.   /*
2.     Create a report showing the first and last names and department
3.     id of all employees whose department is specified.
4.   */
5.
6.   SELECT first_name, last_name, department_id
7.   FROM employees
8.   WHERE department_id IS NOT NULL;
```

Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
1	Steven	King	90
2	Neena	Kochhar	90
3	Lex	De Haan	90
4	Alexander	Hunold	60
5	Bruce	Ernst	60
6	David	Austin	60
7	Valli	Pataballa	60
8	Diana	Lorentz	60
9	Nancy	Greenberg	100
10	Daniel	Faviet	100

Exercise 8: Checking for NULL

5 to 15 minutes

In this exercise, you will practice selecting records with fields that have (or do not have) `NULL` values.

1. Open a new SQL Worksheet in SQL Developer and save it as `null.sql` in `Basic-Selects/Exercises`.
2. Create a report that shows the first and last names and commission percentage of all employees who earn a commission.
3. Create a report that shows the first and last name of all employees who do not report to anybody (who have no manager).

Solution: Basic-Selects/Solutions/null.sql

```
1.   SELECT first_name, last_name, commission_pct
2.   FROM employees
3.   WHERE commission_pct IS NOT NULL;
4.
5.   SELECT first_name, last_name
6.   FROM employees
7.   WHERE manager_id IS NULL;
```

3.10. WHERE and ORDER BY

When using `WHERE` and `ORDER BY` together, the `WHERE` clause must come before the `ORDER BY` clause.

Demo 3.12: Basic-Selects/Demos/where-order-by.sql

```
1.   /*
2.     Create a report showing the first and last name of all
3.     employees whose last names start with a letter in the last
4.     half of the alphabet. Sort by last name in descending order.
5.   */
6.
7.   SELECT first_name, last_name
8.   FROM employees
9.   WHERE last_name >= 'N'
10.  ORDER BY last_name DESC;
```

Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME
1	Eleni	Zlotkey
2	Jennifer	Whalen
3	Matthew	Weiss
4	Alana	Walsh
5	Shanta	Vollman
6	Clara	Vishney
7	Peter	Vargas
8	Jose Manuel	Urman
9	Oliver	Tuvault
10	Peter	Tucker

Exercise 9: Using WHERE and ORDER BY Together

5 to 15 minutes

In this exercise, you will practice writing `SELECT` statements that use both `WHERE` and `ORDER BY`.

1. Open a new SQL Worksheet in SQL Developer and save it as `where-order-by.sql` in `Basic-Selects/Exercises`.
2. Create a report that shows the first and last names and commission percentage of all employees who earn a commission. Show employees with the highest commissions first.
3. Create a report that shows the job title and the minimum and maximum salaries of all jobs with a maximum salary of at least 10000. Sort by maximum salary.

Solution: Basic-Selects/Solutions/where-order-by.sql

```
1.   SELECT first_name, last_name, commission_pct
2.   FROM employees
3.   WHERE commission_pct IS NOT NULL
4.   ORDER BY commission_pct DESC;
5.
6.   SELECT job_title, min_salary, max_salary
7.   FROM jobs
8.   WHERE max_salary >= 10000
9.   ORDER BY max_salary;
```

3.11. Checking Multiple Conditions with Boolean Operators

❖ 3.11.1. AND

`AND` can be used in a `WHERE` clause to find records that match more than one condition.

Demo 3.13: Basic-Selects/Demos/where-and.sql

```
1.   /*
2.     Create a report showing the first and last names,
3.     job id and salary of all sales representatives who
4.     earn at least 10000. Sort by salary.
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE job_id = 'SA_REP'
10.    AND salary >= 10000
11.  ORDER BY salary;
```

Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	Peter	Tucker	SA_REP	10000
2	Janette	King	SA_REP	10000
3	Harrison	Bloom	SA_REP	10000
4	Clara	Vishney	SA_REP	10500
5	Ellen	Abel	SA_REP	11000
6	Lisa	Ozer	SA_REP	11500

❖ 3.11.2. OR

`OR` can be used in a `WHERE` clause to find records that match at least one of several conditions.

Demo 3.14: Basic-Selects/Demos/where-or.sql

```
1.   /*
2.     Create a report showing the first and last names, job id
3.     and salary of all sales representatives (SA_REP) and
4.     sales managers (SA_MAN). Sort by job id.
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE job_id = 'SA_REP' OR job_id = 'SA_MAN'
10.  ORDER BY job_id;
```


Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	Gerald	Cambrault	SA_MAN	11000
2	Alberto	Errazuriz	SA_MAN	12000
3	Karen	Partners	SA_MAN	13500
4	John	Russell	SA_MAN	14000
5	Eleni	Zlotkey	SA_MAN	10500
6	Oliver	Tuvault	SA_REP	7000
7	Janette	King	SA_REP	10000
8	Patrick	Sully	SA_REP	9500
9	Allan	McEwen	SA_REP	9000
10	Lindsey	Smith	SA_REP	8000

❖ 3.11.3. Order of Evaluation

By default, SQL processes `AND` operators before it processes `OR` operators. To illustrate how this works, take a look at the following example.

Demo 3.15: Basic-Selects/Demos/where-and-or-precedence-1.sql

```
1.   /*
2.     Create a report showing the first and last names, job id,
3.     and salary of all employees earning more than 10000 who
4.     are sales representatives or sales managers.
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE job_id = 'SA_REP' OR job_id='SA_MAN'
10.      AND salary > 10000;
11.
12.  -- The query above is equivalent to:
13.  SELECT first_name, last_name, job_id, salary
14.  FROM employees
15.  WHERE job_id = 'SA_REP' OR
16.      (job_id='SA_MAN' AND salary > 10000);
```


Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	John	Russell	SA_MAN	14000
2	Karen	Partners	SA_MAN	13500
3	Alberto	Errazuriz	SA_MAN	12000
4	Gerald	Cambrault	SA_MAN	11000
5	Eleni	Zlotkey	SA_MAN	10500
6	Peter	Tucker	SA_REP	10000
7	David	Bernstein	SA_REP	9500
8	Peter	Hall	SA_REP	9000
9	Christopher	Olsen	SA_REP	8000
10	Nanette	Cambrault	SA_REP	7500

Notice that the report includes several employees who earn less than 10000. This is because this query is including:

1. All sales representatives.
2. Sales managers who earn more than 10000.

This can be fixed by putting the `OR` portion of the clause in parentheses.

Demo 3.16: Basic-Selects/Demos/where-and-or-precedence-2.sql

```
1.   /*
2.     Create a report showing the first and last names, job id,
3.     and salary of all employees earning more than 10000 who
4.     are sales representatives or sales managers.
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE (job_id = 'SA_REP' OR job_id='SA_MAN')
10.       AND salary > 10000;
```

Code Explanation

The parentheses specify that the `OR` portion of the clause should be evaluated first, so the above `SELECT` statement will only return sales reps and managers who earn more than 10000:

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	John	Russell	SA_MAN	14000
2	Karen	Partners	SA_MAN	13500
3	Alberto	Errazuriz	SA_MAN	12000
4	Gerald	Cambrault	SA_MAN	11000
5	Eleni	Zlotkey	SA_MAN	10500
6	Clara	Vishney	SA_REP	10500
7	Lisa	Ozer	SA_REP	11500
8	Ellen	Abel	SA_REP	11000

If only to make the code more readable, it's a good idea to use parentheses whenever the order of precedence might appear ambiguous.

Exercise 10: Writing SELECTs with Multiple Conditions

5 to 15 minutes

In this exercise, you will practice writing `SELECT` statements that filter records based on multiple conditions.

1. Open a new SQL Worksheet in SQL Developer and save it as `multiple-conditions.sql` in `Basic-Selects/Exercises`.
2. Create a report that shows the first and last names, job id, and hire dates of all sales representatives hired in 2006 or later.
3. Create a report that shows the street address, state/province, country id, and postal code of all locations in the United States (US) and the United Kingdom (UK), except for London.

Solution: Basic-Selects/Solutions/multiple-conditions.sql

```
1.   SELECT first_name, last_name, job_id, hire_date
2.   FROM employees
3.   WHERE job_id = 'SA_REP' AND hire_date >= '01-JAN-06'
4.   ORDER BY hire_date;
5.
6.   SELECT street_address, city, state_province, country_id, postal_code
7.   FROM locations
8.   WHERE country_id = 'US' OR
9.       (country_id = 'UK' AND city <> 'London');
```

Code Explanation

Note that the parentheses in the second query are not required, because the `AND` operator takes precedence over the `OR` operator by default. However, they do make the purpose of the query clearer.

3.12. The WHERE Clause and Logical Operator Keywords

The following table shows the word operators used in `WHERE` conditions:

SQL Word Operators

Operator	Description
`BETWEEN`	Returns values in an inclusive range.
`IN`	Returns values in a specified subset.
`LIKE`	Returns values that match a simple pattern.
`NOT`	Negates an operation.

❖ 3.12.1. The BETWEEN Operator

The `BETWEEN` operator is used to check if field values are within a specified inclusive range.

Demo 3.17: Basic-Selects/Demos/where-between.sql

```
1.   /*
2.     Create a report showing the first and last name of all employees
3.     whose last names start with a letter between "J" and "L", meaning
4.     that they would show up in the dictionary after the letter 'J',
5.     but before the letter 'L'. Sort by last name.
6.   */
7.
8.   SELECT first_name, last_name
9.   FROM employees
10.  WHERE last_name BETWEEN 'J' AND 'L'
11.  ORDER BY last_name;
12.
13.  -- The above SELECT statement is the same as the one below.
14.
15.  SELECT first_name, last_name
16.  FROM employees
17.  WHERE last_name >= 'J' AND last_name <= 'L'
18.  ORDER BY last_name;
```

Code Explanation

The above `SELECT` statements will both return the following results:

	FIRST_NAME	LAST_NAME
1	Charles	Johnson
2	Vance	Jones
3	Payam	Kaufling
4	Alexander	Khoo
5	Janette	King
6	Steven	King
7	Neena	Kochhar
8	Sundita	Kumar

Note that a person with the last name of "Jackson" would be included in this report, but a person with the last name of "Lacinski" would not.

❖ 3.12.2. The IN Operator

The `IN` operator is used to check if field values are included in a specified comma-delimited list.

Demo 3.18: Basic-Selects/Demos/where-in.sql

```
1.   /*
2.     Create a report showing the first and last names, job id,
3.     and salary of all sales representatives or sales managers.
4.     Sort by salary, highest to lowest.
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE job_id IN ('SA_REP', 'SA_MAN')
10.  ORDER BY salary DESC;
```

Code Explanation

The above `SELECT` statements will both return the following results:

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	John	Russell	SA_MAN	14000
2	Karen	Partners	SA_MAN	13500
3	Alberto	Errazuriz	SA_MAN	12000
4	Lisa	Ozer	SA_REP	11500
5	Gerald	Cambrault	SA_MAN	11000
6	Ellen	Abel	SA_REP	11000
7	Eleni	Zlotkey	SA_MAN	10500
8	Clara	Vishney	SA_REP	10500
9	Harrison	Bloom	SA_REP	10000
10	Janette	King	SA_REP	10000

❖ 3.12.3. The LIKE Operator

The `LIKE` operator is used to check if field values match a specified pattern.

The Percent Sign (%)

The percent sign (`%`) is used to match any zero or more characters.

Demo 3.19: Basic-Selects/Demos/where-like-1.sql

```
1.   /*
2.     Create a report showing the first and last name of all employees
3.     whose last names begin with the letter 'L'.
4.     Sort by last name.
5.   */
6.
7.   SELECT first_name, last_name
8.   FROM employees
9.   WHERE last_name LIKE 'L%'
10.  ORDER BY last_name;
```

Code Explanation

The above `SELECT` statement will return the following results:

	FIRST_NAME	LAST_NAME
1	Renske	Ladwig
2	James	Landry
3	David	Lee
4	Jack	Livingston
5	Diana	Lorentz

The Underscore (_)

The underscore (_) is used to match any single character.

Demo 3.20: Basic-Selects/Demos/where-like-2.sql

```
1.   /*
2.     Create a report showing the street address, city,
3.     state/province, and country id for all countries that
4.     have an two-letter id starting with 'U'.
5.   */
6.
7.   SELECT street_address, city, state_province, country_id
8.   FROM locations
9.   WHERE country_id LIKE 'U_';
```

Code Explanation

The above SELECT statement will return the following results:

	STREET_ADDRESS	CITY	STATE_PROVINCE	COUNTRY_ID
1	2014 Jabberwocky Rd	Southlake	Texas	US
2	2011 Interiors Blvd	South San Francisco	California	US
3	2007 Zagora St	South Brunswick	New Jersey	US
4	2004 Charade Rd	Seattle	Washington	US
5	8204 Arthur St	London	(null)	UK
6	Magdalen Centre, The Oxford Science Park	Oxford	Oxford	UK
7	9702 Chester Road	Stretford	Manchester	UK

❖ 3.12.4. The NOT Operator

The NOT operator is used to negate an operation.

Demo 3.21: Basic-Selects/Demos/where-not.sql

```
1.   /*
2.     Create a report showing the first and last names, the job id,
3.     and the salary of all employees that are not
4.     sales managers (SA_MAN)or sales reps (SA_REP).
5.   */
6.
7.   SELECT first_name, last_name, job_id, salary
8.   FROM employees
9.   WHERE job_id NOT IN ('SA_MAN.','SA_REP.');
```


Code Explanation

The above SELECT statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	Steven	King	AD_PRES	24000
2	Neena	Kochhar	AD_VP	17000
3	Lex	De Haan	AD_VP	17000
4	Alexander	Hunold	IT_PROG	9000
5	Bruce	Ernst	IT_PROG	6000
6	David	Austin	IT_PROG	4800
7	Valli	Pataballa	IT_PROG	4800
8	Diana	Lorentz	IT_PROG	4200
9	Nancy	Greenberg	FI_MGR	12008
10	Daniel	Faviet	FI_ACCOUNT	9000

Exercise 11: More SELECTs with WHERE

10 to 20 minutes

In this exercise, you will practice writing `SELECT` statements that use `WHERE` conditions with word operators.

1. Open a new SQL Worksheet in SQL Developer and save it as `word-operators.sql` in `Basic-Selects/Exercises`.
2. Create a report that shows the first and last names and hire date of all employees hired between January 1, 2003 and January 1, 2005. Sort by date hired.
3. Create a report showing the street address, city, state/province, and country id for all locations in the United States (`US`), United Kingdom (`UK`), and Japan (`JP`).
4. Create a report that shows the city, country id, and postal code for all postal codes that begin with '0'.
5. Create a report that shows the first and last names and the job id of all employees who have job ids that do not begin with 'SA'.

Solution: Basic-Selects/Solutions/word-operators.sql

```
1.   SELECT first_name, last_name, hire_date
2.   FROM employees
3.   WHERE hire_date BETWEEN '1-Jan-2003' AND '1-Jan-2005'
4.   ORDER BY hire_date;
5.
6.   SELECT street_address, city, state_province, country_id
7.   FROM locations
8.   WHERE country_id IN ('UK', 'US', 'JP')
9.
10.  SELECT city, country_id, postal_code
11.  FROM locations
12.  WHERE postal_code LIKE '0%'
13.  ORDER BY postal_code;
14.
15.  SELECT first_name, last_name, job_id
16.  FROM employees
17.  WHERE job_id NOT LIKE 'SA%'
```


3.13. Limiting Rows

Sometimes you want your query to return a limited number of rows. In Oracle, this is done with the `FETCH` clause. Let's start with an example:

Demo 3.22: Basic-Selects/Demos/fetch-simple.sql

```
1.   SELECT employee_id, first_name, last_name
2.   FROM employees
3.   FETCH FIRST 10 ROWS ONLY;
```

Code Explanation

This query will return the following results:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	100	Steven	King
2	101	Neena	Kochhar
3	102	Lex	De Haan
4	103	Alexander	Hunold
5	104	Bruce	Ernst
6	105	David	Austin
7	106	Valli	Pataballa
8	107	Diana	Lorentz
9	108	Nancy	Greenberg
10	109	Daniel	Faviet

It fetches the first 10 rows as stored in the database.

The following query is similar but with sorting:

Demo 3.23: Basic-Selects/Demos/fetch-with-order-by.sql

```
1.    SELECT employee_id, first_name, last_name
2.    FROM employees
3.    ORDER BY last_name, first_name
4.    FETCH FIRST 10 ROWS ONLY;
```


Code Explanation

This query will return the following results:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	174	Ellen	Abel
2	166	Sundar	Ande
3	130	Mozhe	Atkinson
4	105	David	Austin
5	204	Hermann	Baer
6	116	Shelli	Baida
7	167	Amit	Banda
8	172	Elizabeth	Bates
9	192	Sarah	Bell
10	151	David	Bernstein

Note that the `FETCH` clause comes after the `ORDER BY` clause and gets the first 10 rows after the records have been sorted.

The two queries above both started by fetching the first row. But in some cases, you may want to get a number of rows after skipping a number of rows:

Demo 3.24: Basic-Selects/Demos/fetch-with-offset.sql

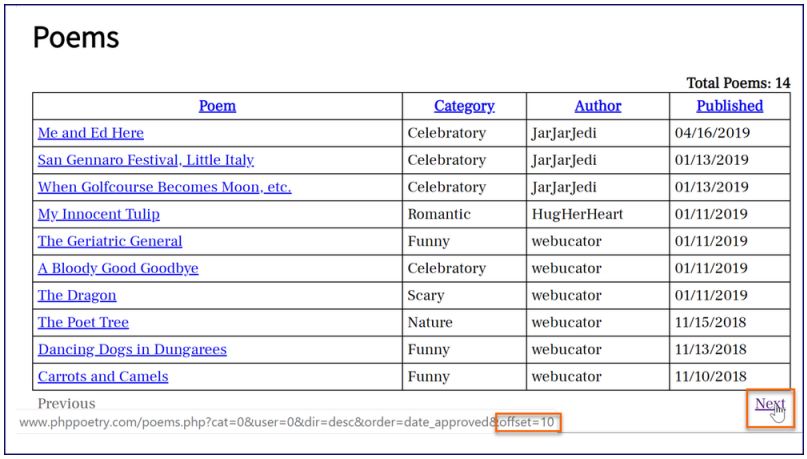
```
1.    SELECT employee_id, first_name, last_name
2.    FROM employees
3.    ORDER BY last_name, first_name
4.    OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

Code Explanation

This query will return the following results:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	129	Laura	Bissot
2	169	Harrison	Bloom
3	185	Alexis	Bull
4	187	Anthony	Cabrio
5	148	Gerald	Cambrault
6	154	Nanette	Cambrault
7	110	John	Chen
8	188	Kelly	Chung
9	119	Karen	Colmenares
10	142	Curtis	Davies

The `OFFSET` is the number of rows to skip. This is useful for pagination. For example, take a look at the following screenshot that depicts a web page showing a listing of poems:



The URL in the status bar shows the next page that will come up when the user clicks on the **Next** link. Notice that it contains "offset=10". This value will be used as the `OFFSET` value in the SQL query that fetches the next 10 rows.

FIRST and NEXT are Synonymous

In the `FETCH` clause, the `FIRST` and `NEXT` keywords are synonymous. `FIRST` is usually used when there is no `OFFSET` and `NEXT` is usually used with an `OFFSET`.

The `FETCH` clause is commonly used when you want to get the first record only:

Demo 3.25: Basic-Selects/Demos/fetch-one-row.sql

```
1.   SELECT employee_id, first_name, last_name, hire_date, job_id
2.   FROM employees
3.   WHERE JOB_ID = 'SA_REP'
4.   ORDER BY hire_date DESC
5.   FETCH FIRST 1 ROW ONLY;
```

Code Explanation

This query will return the following result:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	JOB_ID
1	167	Amit	Banda	21-APR-08	SA_REP

The query fetches the last hired sales representative.

But what if two sales representatives were hired on the same date? The query above will only get one of them. To get them both, we replace `ONLY` with `WITH TIES`:

Demo 3.26: Basic-Selects/Demos/fetch-with-ties.sql

```
1.    SELECT employee_id, first_name, last_name, hire_date, job_id
2.    FROM employees
3.    WHERE JOB_ID = 'SA_REP'
4.    ORDER BY hire_date DESC
5.    FETCH FIRST 1 ROW WITH TIES;
```

Code Explanation

This query will get all the last hired sales representatives, meaning all those who were hired on the same date and after all other sales representatives were hired:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	JOB_ID
1	167	Amit	Banda	21-APR-08	SA_REP
2	173	Sundita	Kumar	21-APR-08	SA_REP

ROWS and ROW are Synonymous

In the `FETCH` clause, the `ROWS` and `ROW` keywords are synonymous. `ROWS` is usually used when fetching multiple rows and `ROW` is used when fetching only one row.

❖ 3.13.1. Fetching a Percent of Records

It is also possible to return a percent of records. For example, to query below gets the first 10% of employees by last name:

Demo 3.27: Basic-Selects/Demos/fetch-percent.sql

```
1.    SELECT employee_id, first_name, last_name, hire_date, job_id
2.    FROM employees
3.    ORDER BY last_name
4.    FETCH FIRST 10 PERCENT ROWS ONLY;
```

Code Explanation

This query will return the following result:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	JOB_ID
1	174	Ellen	Abel	11-MAY-04	SA_REP
2	166	Sundar	Ande	24-MAR-08	SA_REP
3	130	Mozhe	Atkinson	30-OCT-05	ST_CLERK
4	105	David	Austin	25-JUN-05	IT_PROG
5	204	Hermann	Baer	07-JUN-02	PR_REP
6	116	Shelli	Baida	24-DEC-05	PU_CLERK
7	167	Amit	Banda	21-APR-08	SA_REP
8	172	Elizabeth	Bates	24-MAR-07	SA_REP
9	192	Sarah	Bell	04-FEB-04	SH_CLERK
10	151	David	Bernstein	24-MAR-05	SA_REP
11	129	Laura	Bissot	20-AUG-05	ST_CLERK

Note that this will always round up. For example, there are 107 records in the employees table. 10% of 107 is 10.7. So, this query returns 11 records.

Exercise 12: Working with FETCH

10 to 20 minutes

In this exercise, you will practice limiting result sets using the `FETCH` clause.

1. Open a new SQL Worksheet in SQL Developer and save it as `fetch.sql` in `Basic-Selects/Exercises`.
2. Create a report that shows the country id and name and the region id of the **first five** countries ordered by country name.
3. Create a report that shows the country id and name and the region id of the **second five** countries ordered by country name.
4. Create a report that shows first and last names, salary, and job id of the highest paid employee.
5. Create a report that shows first and last names, and salary of the shipping clerk(s) (job id is "SH_CLERK") with the lowest salary.
6. Create a report that shows the highest paid 50% of jobs (by `min_salary`).
7. Create a report that shows the lowest paid 50% of jobs (by `min_salary`).

Solution: Basic-Selects/Solutions/fetch.sql

```
1.   -- First five countries by country name
2.   SELECT country_id, country_name, region_id
3.   FROM countries
4.   ORDER BY country_name
5.   FETCH FIRST 5 ROWS ONLY;
6.
7.   -- Second five countries by country name
8.   SELECT country_id, country_name, region_id
9.   FROM countries
10.  ORDER BY country_name
11.  OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;
12.
13.  -- Highest paid employee
14.  SELECT first_name, last_name, salary, job_id
15.  FROM employees
16.  ORDER BY salary DESC
17.  FETCH FIRST 1 ROW ONLY;
18.
19.  -- Lowest paid Shipping Clerk(s)
20.  SELECT first_name, last_name, salary
21.  FROM employees
22.  WHERE job_id = 'SH_CLERK'
23.  ORDER BY salary ASC
24.  FETCH FIRST 1 ROW WITH TIES;
25.
26.  -- Highest paid 50% of jobs (by min_salary)
27.  SELECT job_id, job_title, min_salary, max_salary
28.  FROM jobs
29.  ORDER BY min_salary DESC
30.  FETCH FIRST 50 PERCENT ROWS ONLY;
31.
32.  -- Lowest paid 50% of jobs (by min_salary)
33.  SELECT job_id, job_title, min_salary, max_salary
34.  FROM jobs
35.  ORDER BY min_salary
36.  FETCH FIRST 50 PERCENT ROWS ONLY;
```


Conclusion

In this lesson, you have learned a lot about creating reports with `SELECT`. However, this is just the tip of the iceberg. `SELECT` statements can get a lot more powerful and, of course, a lot more complicated.

LESSON 4
Oracle SQL Functions

Topics Covered

☑ The DUAL table.

☑ Column aliases.

☑ Calculated fields.

☑ Numeric functions.

☑ Character functions.

☑ Datetime functions.

☑ Null-related functions.

☑ Additional Oracle functions.

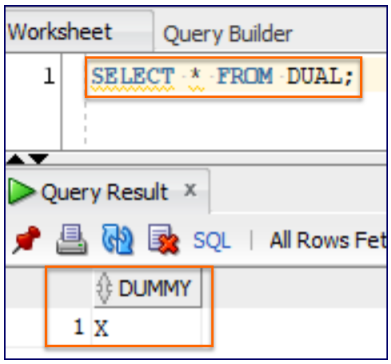
☑ The `TO_CHAR()` function.

Introduction

Oracle includes many "single-row" functions, which return a single result for every row queried. In this lesson, we will cover many of the most commonly used functions.

4.1. The DUAL Table and Column Aliases

The `DUAL` table is a special table automatically created when Oracle is installed. It has a single column named 'DUMMY' and one row of data, in which the value of `DUMMY` is 'X'. The screenshot below shows the results of `SELECT * FROM DUAL;`:



Oracle `SELECT` syntax requires a `FROM` clause, and you can use `DUAL` as a dummy table to select from when the data you are getting doesn't actually exist in a table. It provides a convenient placeholder for outputting the results of arithmetic expressions, Oracle functions, and other expressions that are not derived from table data. In this lesson, we will use `DUAL` for demonstrating how Oracle's SQL functions work. But before that, a quick illustration demonstrating simple math in Oracle:

Demo 4.1: Oracle-SQL-Functions/Demos/dual-simple-math.sql

```
1.    SELECT 5+5, 15-5, 3*4, 18/5
2.    FROM DUAL;
```


Code Explanation

The query above will output the following:

	5+5	15-5	3*4	18/5
1	10	10	12	3.6

Notice that the column names are the math equations themselves. We'll show how to fix that next.

❖ 4.1.1. Column Aliases

Column aliases are used to give columns friendly names. Note that aliases are only used by the query in question. They are not stored in the database.

Demo 4.2: Oracle-SQL-Functions/Demos/dual-simple-math-aliases.sql

```
1.    SELECT 5+5 AS addition, 15-5 AS subtraction,
2.      3*4 AS multiplication, 18/5 AS division
3.    FROM DUAL;
```

Code Explanation

The query above will output the following:

	ADDITION	SUBTRACTION	MULTIPLICATION	DIVISION
1	10	10	12	3.6

If you wish to include one or more spaces in an alias, you must nest the alias in double quotes. For example:

```
SELECT 5+3 AS "Math Problem" FROM DUAL;
```

However, be careful with this, as spaces can create problems when using Oracle with other programming languages.

4.2. Calculated Fields

Calculated fields are fields that do not exist in a table, but are created in the `SELECT` statement. For example, you might want to create `full_name` from `first_name` and `last_name`.

❖ 4.2.1. Concatenation

Concatenation is a fancy word for stringing together different words or characters. In Oracle, the concatenation operator is the double pipe (`||`). Oracle also includes a `CONCAT()` function. We will cover both the concatenation operator and the `CONCAT()` function later in this lesson.

❖ 4.2.2. Mathematical Calculations

Mathematical calculations in SQL are similar to those in other languages.

Mathematical Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Demo 4.3: Oracle-SQL-Functions/Demos/calculations.sql

```
1.   SELECT 5 + 5 AS addition,
2.     5 - 5 AS subtraction,
3.     5 * 5 AS multiplication,
4.     5 / 5 AS division
5.   FROM DUAL;
```


Code Explanation

The above `SELECT` statement will return the following results:

	ADDITION	SUBTRACTION	MULTIPLICATION	DIVISION
1	10	0	25	1

Exercise 13: Calculating Commissions

5 to 15 minutes

Write a query that outputs the following columns:

1. Open a new SQL Worksheet in SQL Developer and save it as `commissions.sql` in `Oracle-SQL-Functions/Exercises`.
2. Each employee's first and last name and commission percentage (`commission_pct`).
3. The commissions each employee who receives a commission percentage would get on 10,000 in sales.
4. The commission percentage as an integer (e.g., `10` instead of `0.1`).

Sort the results by last name and then by first name.

Solution: Oracle-SQL-Functions/Solutions/commissions.sql

```
1.   -- Get commission on 10000 in sales and pct as integer
2.   SELECT first_name, last_name, commission_pct,
3.     10000 * commission_pct AS commission,
4.     commission_pct * 100 AS comm_percent
5.   FROM employees
6.   ORDER BY last_name, first_name;
```

Code Explanation

Notice that when the value of `commission_pct` is `NULL`, the value of the calculated fields are also `NULL`.

*

4.3. ROW_NUMBER()

The `ROW_NUMBER()` function is one of many analytic functions[1] available in Oracle. While we are not going to cover analytic functions, we do want to show you how to accurately include the row number in a query:

Demo 4.4: Oracle-SQL-Functions/Demos/row-number.sql

```
1.   SELECT employee_id, first_name, last_name, hire_date,
2.     ROW_NUMBER() OVER (ORDER BY hire_date) AS row_num
3.   FROM employees
4.   ORDER BY row_num;
5.
6.   -- Same as above, but with main ORDER BY clause removed
7.   SELECT employee_id, first_name, last_name, hire_date,
8.     ROW_NUMBER() OVER (ORDER BY hire_date) AS row_num
9.   FROM employees;
10.
11.  -- Same as above, but ordering by a different field
12.  SELECT employee_id, first_name, last_name, hire_date,
13.    ROW_NUMBER() OVER (ORDER BY hire_date) AS row_num
14.  FROM employees
15.  ORDER BY last_name;
```

1. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Analytic-Functions.html

Code Explanation

This first query will return the following results (first 10 rows shown):

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	ROW_NUM
1	102	Lex	De Haan	13-JAN-01	1
2	206	William	Gietz	07-JUN-02	2
3	204	Hermann	Baer	07-JUN-02	3
4	203	Susan	Mavris	07-JUN-02	4
5	205	Shelley	Higgins	07-JUN-02	5
6	109	Daniel	Faviet	16-AUG-02	6
7	108	Nancy	Greenberg	17-AUG-02	7
8	114	Den	Raphaely	07-DEC-02	8
9	122	Payam	Kaufling	01-MAY-03	9
10	115	Alexander	Khoo	18-MAY-03	10

Notice that we order by `row_num` in the main `ORDER BY` clause. Technically, we do not have to do this: if we do not include a main `ORDER BY` clause, the results will be sorted according to the order specified in the last `ROW_NUMBER()` calculation in the `SELECT` clause. You can see this by running the second query, which does not include the main `ORDER BY` clause, but returns the same results.

Run the third query to see that using a different `ORDER BY` clause will result in unordered row numbers (first 10 rows shown):

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	ROW_NUM
1	174	Ellen	Abel	11-MAY-04	20
2	166	Sundar	Ande	24-MAR-08	105
3	130	Mozhe	Atkinson	30-OCT-05	50
4	105	David	Austin	25-JUN-05	38
5	204	Hermann	Baer	07-JUN-02	3
6	116	Shelli	Baida	24-DEC-05	53
7	167	Amit	Banda	21-APR-08	106
8	172	Elizabeth	Bates	24-MAR-07	84
9	192	Sarah	Bell	04-FEB-04	17
10	151	David	Bernstein	24-MAR-05	35

To learn more about what you can do with `ROW_NUMBER()`, see `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/ROW_NUMBER.html`.

ROWNUM Pseudocolumn

There is also a `ROWNUM` pseudocolumn that is commonly used for returning row numbers, but the `ROW_NUMBER()` provides better performance. See `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/ROWNUM-Pseudocolumn.html` for details.

4.4. Numeric Functions

Numeric functions take one or more numeric arguments and return a numeric value.

❖ 4.4.1. ABS(), POWER(), and SQRT()

There are a multitude of basic mathematical functions, including:

1. `ABS(num)` returns the absolute value of `num`.
2. `POWER(num1, num2)` returns `num1` raised to the power of `num2`.
3. `SQRT(num)` returns the square root of `num`.

Sample code is shown below:

Demo 4.5: Oracle-SQL-Functions/Demos/basic-math-functions.sql

```
1.   SELECT ABS(-5), ABS(5),
2.     POWER(4,2) AS four_squared,
3.     POWER(4,3) AS four_cubed,
4.     SQRT(4) AS sqrt_of_four
5.   FROM DUAL;
```

Code Explanation

This query will return the following results:

	ABS(-5)	ABS(5)	FOUR_SQUARED	FOUR_CUBED	SQRT_OF_FOUR
1	5	5	16	64	2

❖ 4.4.2. CEIL(), FLOOR(), and ROUND()

1. `CEIL(num)` rounds `num` up to the nearest integer.
2. `FLOOR(num)` rounds `num` down to the nearest integer.
3. `ROUND(num)` rounds `num` according to normal mathematical rounding rules.

Demo 4.6: Oracle-SQL-Functions/Demos/ceil-floor-round.sql

```
1.   -- CEIL(), FLOOR(), ROUND()
2.   SELECT CEIL(5.5) AS round_up,
3.     FLOOR(5.5) AS round_down,
4.     ROUND(5.5) AS round
5.   FROM DUAL;
6.
7.   -- CEIL(), FLOOR(), ROUND(): negative numbers
8.   SELECT CEIL(-5.5) AS round_up,
9.     FLOOR(-5.5) AS round_down,
10.    ROUND(-5.5) AS round
11.  FROM DUAL;
```


Code Explanation

This first query will return the following results:

	ROUND_UP	ROUND_DOWN	ROUND
1	6	5	6

This second query will return the following results:

	ROUND_UP	ROUND_DOWN	ROUND
1	-5	-6	-6

❖ 4.4.3. ROUND(num1, num2) and TRUNC(num1, num2)

`ROUND(num1, num2)`, with a a second argument, rounds to a specified precision:

1. If the second argument is `0`, it rounds to the integer. For example, `ROUND(55.5555, 0)` rounds to `56`. As `0` is the default, this is the same as `ROUND(55.5555)`.

2. If the second argument is less than `0`, it rounds the first argument to a place right of the decimal. For example, `ROUND(55.5555, -1)` returns `60`, `ROUND(55.5555, -2)` returns `100`, etc.
3. If the second argument is greater than `0`, it rounds the first argument to a place left of the decimal. For example, `ROUND(55.5555, 1)` returns `55.6`, `ROUND(55.5555, 2)` returns `55.56`, etc.

`TRUNC(num)` works similarly to `ROUND(num)`, but instead of rounding `num`, it *truncates* `num`, meaning that it just chops off numbers from the end of the number without rounding:

1. If the second argument is `0`, it truncates the decimal portion of the number. For example, `TRUNC(55.5555, 0)` returns `55`. As `0` is the default, this is the same as `TRUNC(55.5555)`.
2. If the second argument is less than `0`, it truncates the first argument to a place right of the decimal. For example, `TRUNC(55.5555, -1)` returns `50`, `TRUNC(55.5555, -2)` returns `0`, etc.
3. If the second argument is greater than `0`, it truncates the first argument to a place left of the decimal. For example, `TRUNC(55.5555, 1)` returns `55.5`, `TRUNC(55.5555, 2)` returns `55.55`, etc.

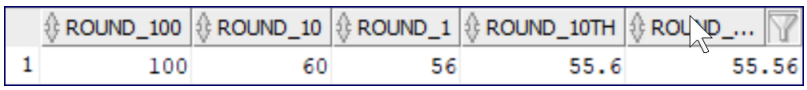
The code below illustrates this:

Demo 4.7: Oracle-SQL-Functions/Demos/round-and-trunc.sql

```
1.   -- ROUND() to place
2.   SELECT ROUND(55.5555, -2) AS round_100,
3.     ROUND(55.5555, -1) AS round_10,
4.     ROUND(55.5555, 0) AS round_1, --default
5.     ROUND(55.5555, 1) AS round_10th,
6.     ROUND(55.5555, 2) AS round_100th
7.   FROM DUAL;
8.
9.   -- TRUNC()
10.  SELECT TRUNC(55.5555, -2) AS trunc_100,
11.    TRUNC(55.5555, -1) AS trunc_10,
12.    TRUNC(55.5555, 0) AS trunc_1, --default
13.    TRUNC(55.5555, 1) AS trunc_10th,
14.    TRUNC(55.5555, 2) AS trunc_100th,
15.    TRUNC(-55.5555, 1) AS trunc_neg_10th,
16.    ROUND(-55.5555, 1) AS round_neg_10th
17.  FROM DUAL;
```

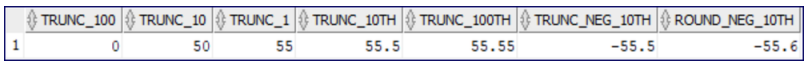
Code Explanation

This first query will return the following results:



	ROUND_100	ROUND_10	ROUND_1	ROUND_10TH	ROU ND_...
1	100	60	56	55.6	55.56

This second query will return the following results:



	TRUNC_100	TRUNC_10	TRUNC_1	TRUNC_10TH	TRUNC_100TH	TRUNC_NEG_10TH	ROUND_NEG_10TH
1	0	50	55	55.5	55.55	-55.5	-55.6

❖ 4.4.4. MOD()

The `MOD()` function is used to get the remainder after division. For example:

1. 5 divided by 2 equals 2 with a remainder of 1.
2. 8 divided by 3 equals 2 with a remainder of 2.
3. 10 divided by 4 equals 2 with a remainder of 2.
4. 99 divided by 25 equals 3 with a remainder of 24.
5. 100 divided by 25 equals 4 with a remainder of 0.

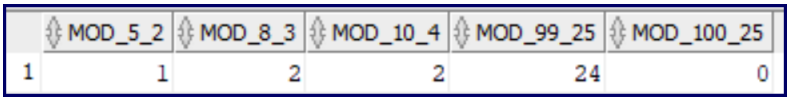
These examples are shown in the code below:

Demo 4.8: Oracle-SQL-Functions/Demos/mod.sql

```
1.   SELECT MOD(5,2) AS mod_5_2,
2.     MOD(8,3) AS mod_8_3,
3.     MOD(10,4) AS mod_10_4,
4.     MOD(99,25) AS mod_99_25,
5.     MOD(100,25) AS mod_100_25
6.   FROM DUAL;
```

Code Explanation

This query will return the following results:



	MOD_5_2	MOD_8_3	MOD_10_4	MOD_99_25	MOD_100_25
1	1	2	2	24	0

At first glance, you might wonder when you would ever user the `MOD()` function, but it actually can be incredibly handy.

MOD() Use Case: Odd and Even Rows

By combining `MOD()` with `ROW_NUMBER()`, you can assign every nth row to a bucket. The query below shows how to do this to break rows into even (`0`) and odd (`1`) with this function: `MOD(ROW_NUMBER() OVER (ORDER BY job_title), 2)`, which divides the row number by `2` and returns the remainder. If the remainder is `1`, then the row is odd. If the remainder is `0`, the row is even:

1. Row `1` is odd because `1/2` equals `0` with a remainder of `1`.
2. Row `2` is even because `2/2` equals `1` with a remainder of `0`.
3. Row `3` is odd because `3/2` equals `1` with a remainder of `1`.
4. Row `4` is even because `4/2` equals `2` with a remainder of `0`.
5. etc.

Demo 4.9: Oracle-SQL-Functions/Demos/mod-odd-even.sql

```
1.   -- Divide jobs into 2 buckets
2.   SELECT job_id, job_title,
3.     ROW_NUMBER() OVER (ORDER BY job_title) AS row_num,
4.     MOD(ROW_NUMBER() OVER (ORDER BY job_title), 2) AS bucket
5.   FROM jobs;
```


Code Explanation

This query will return the following results (first 10 rows shown):

	JOB_ID	JOB_TITLE	ROW_NUM	BUCKET
1	FI_ACCOUNT	Accountant	1	1
2	AC_MGR	Accounting Manager	2	0
3	AD_ASST	Administration Assistant	3	1
4	AD_VP	Administration Vice President	4	0
5	FI_MGR	Finance Manager	5	1
6	HR_REP	Human Resources Representative	6	0
7	MK_MAN	Marketing Manager	7	1
8	MK_REP	Marketing Representative	8	0
9	AD_PRES	President	9	1
10	IT_PROG	Programmer	10	0

Exercise 14: Using MOD()

10 to 20 minutes

The sales managers have a new plan to foster collaboration and competition. They want all the sales reps divided up into three teams. Each team will work together for the next month and the team that makes the most sales will get a prize. To make sure the teams are fair, the sales managers want the teams to be made up of sales reps with equal experience based on the date they were hired. They have asked you to create a report that sorts the sales reps by hire date and assigns them to teams, which for now we will call teams 0, 1, and 2.

Before you can say "modulus," another experienced SQL developer says she has finished the report and shows you the first 10 rows:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	ROW_NUM	TEAM
1	156	Janette	King	30-JAN-04	1	1
2	157	Patrick	Sully	04-MAR-04	2	2
3	174	Ellen	Abel	11-MAY-04	3	0
4	158	Allan	McEwen	01-AUG-04	4	1
5	150	Peter	Tucker	30-JAN-05	5	2
6	159	Lindsey	Smith	10-MAR-05	6	0
7	168	Lisa	Ozer	11-MAR-05	7	1
8	175	Alyssa	Hutton	19-MAR-05	8	2
9	151	David	Bernstein	24-MAR-05	9	0
10	152	Peter	Hall	20-AUG-05	10	1

Can you write the query that produced this report?

Open a new SQL Worksheet in SQL Developer and save it as `mod.sql` in `Oracle-SQL-Functions/Exercises`.

Solution: Oracle-SQL-Functions/Solutions/mod.sql

```
1.   SELECT employee_id, first_name, last_name, hire_date,
2.     ROW_NUMBER() OVER (ORDER BY hire_date) AS row_num,
3.     MOD(ROW_NUMBER() OVER (ORDER BY hire_date), 3) AS team
4.   FROM employees
5.   WHERE job_id = 'SA_REP'
6.   ORDER BY hire_date;
```

4.5. Character Functions Returning Character Values

❖ 4.5.1. TO_CHAR(number, format_model)

The `TO_CHAR(number, format_model)` function converts a number to a string representation of that number based on the specified `format_model`.

Number Format Models

The most common number format models are:

- `9` - digits. `TO_CHAR(1000, '999999')` returns `'1000'`. Note that you must have at least as many places in the `format_model` as digits in the number.
- `0` - digits with leading zeroes if there are more places in the `format_model` than digits in the number. `TO_CHAR(1000, '000000')` returns `'001000'`
- `,` - comma separator. `TO_CHAR(1000, '999,999')` returns `'1,000'`
- `.` - period separator. `TO_CHAR(1000, '999,999.99')` returns `'1,000.00'`
- `$` - leading dollar sign. `TO_CHAR(1000, '$999,999.99')` returns `'$1,000.00'`

For additional documentation on format models, see `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Format-Models.html`.

Open `Oracle-SQL-Functions/Demos/to-char-numbers.sql` for an example query showing the `TO_CHAR(number, format_model)` function.

❖ 4.5.2. CONCAT()

As mentioned earlier, *concatenation* is a fancy word for stringing together different words or characters. Oracle's `CONCAT()` function takes two arguments and combines them into a single *string* (a sequence of characters). For example, `CONCAT('John', 'Wayne')` would return 'JohnWayne'. Note that there is no space between the two strings. If you want a space, you must explicitly include it: `CONCAT('John ', 'Wayne')` would return 'John Wayne'.

As covered earlier in this lesson, in addition to the `CONCAT()` function, Oracle includes a concatenation operator: the double pipe (`||`). The code below shows how it compares to the `CONCAT()` function:

Demo 4.10: Oracle-SQL-Functions/Demos/concat.sql

```
1.   SELECT CONCAT('John', 'Wayne'),
2.     CONCAT('John ', 'Wayne'),
3.     'John' || 'Wayne',
4.     'John' || ' ' || 'Wayne'
5.   FROM DUAL;
```


Code Explanation

This query will return the following results:

	CONCAT('JOHN','WAYNE')	CONCAT('JOHN','WAYNE')_1	'JOHN'||'WAYNE'	'JOHN'||''||'WAYNE'
1	JohnWayne	John Wayne	JohnWayne	John Wayne

It's a little silly to concatenate literal strings like this. As in the following demo, usually we're concatenating interpreted values (e.g., from columns) with each other and/or with literal strings.

A big advantage of using the concatenation operator is that it is easier to concatenate more than two strings. Consider the following queries:

Demo 4.11: Oracle-SQL-Functions/Demos/concat-multiple-strings.sql

```
1.   SELECT street_address || ', ' ||
2.     city || ', ' ||
3.     state_province || ' ' ||
4.     postal_code AS address
5.   FROM locations
6.   WHERE state_province IS NOT NULL;
7.
8.   -- The same query using the CONCAT() function
9.   SELECT CONCAT(
10.          CONCAT(
11.            CONCAT(
12.              CONCAT(
13.                CONCAT(
14.                  CONCAT(street_address, ', '),
15.                    city),
16.                  ', '),
17.                state_province),
18.              ' '),
19.            postal_code) AS address
20.  FROM locations
21.  WHERE state_province IS NOT NULL;
```


Code Explanation

Both queries will return the same results (first 10 rows shown):

	ADDRESS
1	2017 Shinjuku-ku, Tokyo, Tokyo Prefecture 1689
2	2014 Jabberwocky Rd, Southlake, Texas 26192
3	2011 Interiors Blvd, South San Francisco, California 99236
4	2007 Zagora St, South Brunswick, New Jersey 50090
5	2004 Charade Rd, Seattle, Washington 98199
6	147 Spadina Ave, Toronto, Ontario M5V 2L7
7	6092 Boxwood St, Whitehorse, Yukon YSW 9T2
8	1298 Vileparle (E), Bombay, Maharashtra 490231
9	12-98 Victoria Street, Sydney, New South Wales 2901
10	Magdalen Centre, The Oxford Science Park, Oxford, Oxford OX9 9ZB

Notice how much more complicated it is to use the `CONCAT()` function.

Note that non-character data will be converted to character data when concatenating.

Exercise 15: Concatenation

10 to 15 minutes

Open a new SQL Worksheet in SQL Developer and save it as `concat.sql` in `Oracle-SQL-Functions/Exercises`.

Using the concatenation operator, write a query that returns the following string for each employee:


```
Steven King was hired on 17-JUN-03.
```

Challenge

In the same file, try writing the same query using the `CONCAT()` function. Note that this an academic exercise. You would be unlikely to do it this way in the real world, but it will give you some practice nesting functions within functions. Try to indent the code to make it as readable as possible.

Solution: Oracle-SQL-Functions/Solutions/concat.sql

```
1.   SELECT first_name || ' ' ||
2.     last_name  || ' was hired on ' || hire_date || '.' AS sentence
3.   FROM employees;
```

Challenge Solution: Oracle-SQL-Functions/Solutions/concat-challenge.sql

```
1.   SELECT CONCAT(
2.            CONCAT(
3.              CONCAT(
4.                CONCAT(
5.                  CONCAT(first_name, ' '),
6.                    last_name),
7.                  ' was hired on '),
8.                hire_date),
9.              '.') AS sentence
10.  FROM employees;
```


4.6. More Character Functions Returning Character Values

❖ 4.6.1. LOWER(), UPPER(), and INITCAP()

The `LOWER()`, `UPPER()`, and `INITCAP()` functions are used to change the case of a string:

1. `LOWER()` makes all the letters lowercase: `LOWER('Barty Crouch, Sr.')` returns `barty crouch, sr.`
2. `UPPER()` makes all the letters uppercase: `UPPER('Barty Crouch, Sr.')` returns `BARTY CROUCH, SR.`
3. `INITCAP()` makes the first letter of each word uppercase and the rest of the letters lowercase: `INITCAP('barty crouch, SR.')` returns `Barty Crouch, Sr.`

Open `Oracle-SQL-Functions/Demos/strings.sql` for an example query showing the `LOWER()`, `UPPER()`, and `INITCAP()` functions.

❖ 4.6.2. LPAD() and RPAD()

The `LPAD()` and `RPAD()` functions are used to add padding to the left and right of a string. They are often used for lining up content within a column. For example, `LPAD('Snape', 10)` would add five spaces to the left of 'Snape' making the whole length of the new string 10.

Both functions can take a third argument: the character to use for padding. By default that is a space, so:

```
LPAD('Snape', 10) -- returns '     Snape'

LPAD('Snape', 10, ' ') -- also returns '     Snape'

LPAD('Snape', 10, '-') -- returns '-----Snape'
```

Note that if the string in the first argument has more characters than those allotted by the second argument, it will be truncated:

```
LPAD('Severus Snape',10) -- returns 'Severus Sn'
```


Open `Oracle-SQL-Functions/Demos/strings.sql` for an example query showing the `LPAD()` and `RPAD()` functions.

❖ 4.6.3. TRIM(), LTRIM(), and RTRIM()

The `TRIM()`, `LTRIM()`, and `RTRIM()` functions are used to remove leading and trailing spaces from a string:

```
TRIM('   Harry   ') -- returns 'Harry'

LTRIM('   Harry   ') -- returns 'Harry   '

RTRIM('   Harry   ') -- returns '   Harry'
```

`LTRIM()` and `RTRIM()`, but not `TRIM()`, take a second argument specifying which characters to trim. For example, say you have a word surrounded by square brackets and spaces: `' [Harry] '`:

```
LTRIM(' [Harry] ', ' []') -- returns 'Harry] '

RTRIM(' [Harry] ', ' []') -- returns ' [Harry'
```

And to combine the two:

```
RTRIM(LTRIM(' [Harry] ', ' []') ,' []') -- returns 'Harry'
```

Open `Oracle-SQL-Functions/Demos/strings.sql` for an example query showing the `TRIM()`, `LTRIM()`, and `RTRIM()` functions.

The `TRIM()` function has some additional options, which you can see at `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/TRIM.html`.

❖ 4.6.4. REPLACE() and SUBSTR()

The `REPLACE(str1, str2, str3)` function is used to replace one string (`str2`) with another (`str3`) in a string (`str1`). For example, `REPLACE('I like this house.', 'this', 'that')` returns 'I like that house.'

The `SUBSTR(string, start, length)` function returns a `length`-character substring of `string` starting at position `start`. For example, `substr('I like this house.', 3, 4)` returns 'like'.

If the `length` argument is not included, it returns the rest of the string starting from `start`.

Open `Oracle-SQL-Functions/Demos/strings.sql` for an example query showing the `REPLACE()` and `SUBSTR()` functions.

REGEXP_REPLACE() and REGEXP_SUBSTR()

Regular expressions are used to do pattern matching in many programming languages, including Java, Python, PHP, JavaScript, C, and C++.

Oracle includes regular expression functions as well, including `REGEXP_REPLACE()` and `REGEXP_SUBSTR()`. These functions are useful when you are searching for a substring based on a pattern: for example, you might be searching for a substring surrounded by one or more spaces or including a colon. Regular expressions are beyond our scope, but you should know that Oracle supports them. See:

1. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/REGEXP_REPLACE.html
2. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/REGEXP_SUBSTR.html

4.7. Character Functions Returning Number Values

❖ 4.7.1. TO_NUMBER(string)

The `TO_NUMBER()` function converts a string to a number:

```
TO_NUMBER('42') -- returns 42 as a number
```

The function takes an optional second parameter: a number format model to help with the conversion.

❖ 4.7.2. INSTR() and LENGTH()

The `INSTR(string, substring, start, n)` function starts searching with the character at position `start` and returns the position of the nth occurrence of `substring` within `string`. The `start` and `n` parameters both default to `1`. Take, for example, the string 'Webucator educates.' The substring 'cat' shows up twice in that string, once starting at position `5` and again starting at position `14`:

```
INSTR('Webucator educates.', 'cat') -- returns 5

INSTR('Webucator educates.', 'cat', 6) -- returns 14

INSTR('Webucator educates.', 'cat', 1, 2) -- returns 14
```

The `LENGTH(string)` function returns the number of characters in `string`. For example, `LENGTH('Dumbledore')` returns `10`.

Open `Oracle-SQL-Functions/Demos/char-functions-returning-numbers.sql` for an example query showing the `INSTR()` and `LENGTH()` functions.

REGEXP_INSTR() and REGEXP_COUNT()

The `REGEXP_INSTR()` and `REGEXP_COUNT()` are similar to `INSTR()` and `LENGTH()`, but take patterns instead of substrings. See:

1. `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/REGEXP_INSTR.html`
2. `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/REGEXP_COUNT.html`

4.8. Datetime Functions

❖ 4.8.1. CURRENT_DATE, CURRENT_TIMESTAMP, SYSDATE, and SYSTIMESTAMP

Parameter-less Functions

There are some built-in Oracle SQL functions that do not take any parameters. When calling these functions, you do not use parentheses. The four functions covered in this section are examples of such functions.

The `CURRENT_DATE` and `CURRENT_TIMESTAMP` functions return the current date and current date and time, respectively, in the user's timezone.

The `SYSDATE` and `SYSTIMESTAMP` functions return the current date and current date and time, respectively, in the timezone set on the operating system on which the database is running.

Open `Oracle-SQL-Functions/Demos/datetime.sql` for an example query showing these functions.

❖ 4.8.2. TO_DATE()

The `TO_DATE(date_string, format_model)` function converts `date_string` represented according to `format_model` to a date. For example, `TO_DATE('07-04-1776', 'MM-DD-YYYY')` returns July

4, 1776 as a date, so that you can then perform date operations on it. For example, the American Revolution began on April 19, 1775 and ended September 3, 1783. To find out how many days the war lasted, you must first create the two dates and then subtract the start date from the end date:

```
-- Start of War:
TO_DATE('04-19-1775', 'MM-DD-YYYY')

-- End of War:
TO_DATE('09-03-1783', 'MM-DD-YYYY')

-- Days of War:
TO_DATE('09-03-1783', 'MM-DD-YYYY')
  - TO_DATE('04-19-1775', 'MM-DD-YYYY')
```

❖ 4.8.3. TO_CHAR(datetime, format_model)

The inverse function of `TO_DATE()` is `TO_CHAR(datetime, format_model)`, which converts a date to a string representation of that date. For example, the following will return the current day of the week:


```
TO_CHAR(CURRENT_DATE, 'Day')
```

In some cases, you may want to create a date from a string and then use `TO_CHAR()` to get that date back in a different format. For example:

```
SELECT 'Independence was declared on a ' ||
  TO_CHAR(
    TO_DATE('07-04-1776', 'MM-DD-YYYY'),
    'Day'
  ) AS independence_day
FROM DUAL;
```

This first converts the string '07-04-1776' into a date and then uses `TO_CHAR()` to get the day of the week of that date.

Open `Oracle-SQL-Functions/Demos/datetime.sql` for an example query showing the `TO_DATE()` function.

Date Format Models

The most common date format models are:

- `YYYY` - four-digit year.
- `MM` - one or two-digit month (with or without leading zeroes).
- `MON` - abbreviated month (e.g., 'JAN', 'FEB', etc.).
- `MONTH` - full month (e.g., 'January', 'February', etc.).
- `D` - day of week as a number (e.g., 1-7)
- `DD` - one or two-digit day (with or without leading zeroes).
- `DY` - abbreviated day (e.g., 'MON', 'TUE', etc.).
- `DAY` - full day (e.g., 'Monday', 'Tuesday', etc.).

For additional documentation on format models, see `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Format-Models.html`.

❖ 4.8.4. ROUND() and TRUNC()

The `ROUND(date, format_model)` function returns `date` rounded according to the `format_model`. For example, `ROUND(CURRENT_DATE, 'YYYY')` will return (as a date) January 1 of the current year (for dates 1/1 through 6/30) or of the next year (for dates 7/1 through 12/31).

The `TRUNC(date, format_model)` function returns `date` truncated according to the `format_model`. For example, `TRUNC(CURRENT_DATE, 'YYYY')` will return (as a date) January 1 of the current year.

Open `Oracle-SQL-Functions/Demos/datetime.sql` for an example query showing the `ROUND()` and `TRUNC()` functions with dates.

For full documentation on the date format models you can use with `ROUND()` and `TRUNC()`, see `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/ROUND-and-TRUNC-Date-Functions.html`.

Exercise 16: Dates

15 to 25 minutes

1. Open a new SQL Worksheet in SQL Developer and save it as `datetime.sql` in `Oracle-SQL-Functions/Exercises`.
2. Write a query that shows how long each employee has been working for the company. Assume that all employees are still working today. Show the results in both:
    A. Days rounded to an integer in the format "5010 days".
    B. In years (using 365.25 days as the length of the year) rounded to the 100th place in the format "13.72 years".
3. As we saw earlier, the American Revolution began on April 19, 1775 and ended September 3, 1783. Using the date functions you have learned and the `MOD()` function covered earlier, write a query that outputs:
    A. The number of years the war lasted (using 365.25 days as the length of the year). For an alias, use `years_of_war`.
    B. The length fo the war in years and days as two separate fields: `years_only` and `and_days`.
4. Write a query that shows the day of the week that each person was hired on. Sort by the day of the week as the occur (e.g, Sunday, Monday, Tuesday,...). You may find it helpful to review the date format models.

Solution: Oracle-SQL-Functions/Solutions/datetime.sql

```
1.   --How long employed
2.   SELECT first_name, last_name, hire_date,
3.     ROUND(current_date - hire_date)
4.       || ' days' AS days_employed,
5.     ROUND((current_date - hire_date) / 365.25, 2)
6.       || ' years' AS years_employed
7.   FROM employees;
8.
9.   -- Revolutionary War: April 19, 1775 - September 3, 1783
10.  SELECT TO_DATE('04-19-1775', 'MM-DD-YYYY') AS war_starts,
11.    TO_DATE('09-03-1783', 'MM-DD-YYYY') AS war_ends,
12.    TO_DATE('09-03-1783', 'MM-DD-YYYY')
13.      - TO_DATE('04-19-1775', 'MM-DD-YYYY') AS days_of_war,
14.    (TO_DATE('09-03-1783', 'MM-DD-YYYY')
15.      - TO_DATE('04-19-1775', 'MM-DD-YYYY')) / 365.25 AS years_of_war,
16.    TRUNC(
17.      (TO_DATE('09-03-1783', 'MM-DD-YYYY')
18.      - TO_DATE('04-19-1775', 'MM-DD-YYYY')) / 365.25
19.    ) AS years_only,
20.    MOD(TO_DATE('09-03-1783', 'MM-DD-YYYY')
21.      - TO_DATE('04-19-1775', 'MM-DD-YYYY'), 365.25)
22.        AS and_days
23.  FROM DUAL;
24.
25.  SELECT first_name, last_name, TO_CHAR(hire_date, 'Day') AS hire_day
26.  FROM employees
27.  ORDER BY TO_CHAR(hire_date, 'D');
```

Code Explanation

This American-Revolution query should return the following values:

1. `years_of_war`: `8.375085557837097878165639972621492128 68`
2. `years_only`: `8`
3. `and_days`: `137`

4.9. SQL*Plus column Command

Oracle comes with a built-in interactive query tool called *SQL*Plus*. SQL*Plus includes some commands that are not part of standard SQL or PL/SQL. These commands are available in SQL Developer as well.

SQL*Plus's `column` command allows you to format a column generated by a `SELECT` statement on the report. The command can be abbreviated `col`. Consider the following example:

```
col full_name format a15
col salary format $999,999.99
SELECT first_name || ' ' || last_name as full_name, salary
FROM employees;
```

The width of the the `full_name` column will be fixed as 15 alphanumeric characters and the `salary` column will be formatted with a dollar sign, 6 digits to the left of the decimal point, comma separating thousands and a decimal point followed by two digits.

The output will look like this:

```
FULL_NAME             SALARY
--------------- ------------
Steven King       $24,000.00
Neena Kochhar     $17,000.00
Lex De Haan       $17,000.00
Alexander Hunol    $9,000.00
d

Bruce Ernst        $6,000.00
David Austin       $4,800.00
Valli Pataballa    $4,800.00
Diana Lorentz      $4,200.00
Nancy Greenberg   $12,008.00
```

Notice that Alexander Hunold's name is wrapped to the next line. That's because it is longer than 15 characters.

4.10. NULL-Related Functions

Remember that when a field in a row has no value, it is said to be `NULL`. Oracle includes special functions to deal with `NULL` values.

These functions are often used to replace possible `NULL` values with more meaningful values in reports.

❖ 4.10.1. COALESCE()

The `COALESCE(expr1, expr2, ...)` function takes two or more expressions and returns the first expression that does not evaluate to `NULL`. For example:

```
COALESCE(NULL, 'foo', 'bar'), -- returns foo

COALESCE('1', 'foo', 'bar'), -- returns '1'
```


❖ 4.10.2. NVL()

The `NVL(expr1, expr2)` function is almost identical to `COALESCE()`, but it only allows for two expressions. If `expr1` is not `NULL` then it is returned. Otherwise, `expr2` is returned. For example:

```
NVL(NULL, 'bar'), -- returns bar

NVL('foo', NULL), -- returns foo
```

❖ 4.10.3. NVL2()

The `NVL2(expr1, expr2, expr3)` function takes three expressions. If `expr1` is not `NULL`, `expr2` is returns. Otherwise, `expr3` is returned. For example:

```
NVL2(NULL, 'foo', 'bar'), -- returns bar

NVL2(1, 'foo', 'bar') -- returns foo
```

Exercise 17: NULL Functions

15 to 25 minutes

Open a new SQL Worksheet in SQL Developer and save it as `null-functions.sql` in `Oracle-SQL-Functions/Exercises`.

Write a query that outputs the following columns:

1. Each employee's first and last name and commission percentage (`commission_pct`).
2. The commissions each employee would get on 10,000 in sales. For those employees who don't receive a commission percentage, output `0`.
3. The commission percentage as an integer (e.g., `10` instead of `0.1`). For those employees who don't receive a commission percentage, output `0`.

Write a second query that outputs the following columns:

1. Each employee's id, and first and last name.
2. The employee's manager's id. If the employee has no manager, output the employee's own id.
3. The employee's manager's id. If the employee has no manager, output `0`.
4. "Has Boss" for employees who have managers and "Has No Boss" for those who don't.

Solution: Oracle-SQL-Functions/Solutions/null-functions.sql

```
1.   -- Get commission on 10000 in sales and pct as integer
2.   SELECT first_name, last_name, commission_pct,
3.     COALESCE(10000 * commission_pct, 0) AS commission,
4.     COALESCE(commission_pct * 100, 0) AS comm_percent
5.   FROM employees;
6.
7.   -- Get manager_id in different ways
8.   SELECT employee_id, first_name, last_name,
9.     COALESCE(manager_id, employee_id) AS boss_coalesce,
10.    NVL(manager_id, 0) AS boss_nvl,
11.    NVL2(manager_id, 'Has Boss', 'Has No Boss') AS boss_nvl2
12.  FROM employees;
```

4.11. Other Functions

❖ 4.11.1. DECODE()

The `DECODE()` function takes:

1. An expression to evaluate.
2. One or more expression pairs. If the first value in the pair is equal to the expression to evaluate, the function returns the second value in the pair. If the first value in the pair is **not** equal to the expression to evaluate, the function moves on to the next pair.
3. An optional default value, which is returned if none of the first values of the expression pairs is equal to the expression to evaluate. If none of the first values of the expression pairs is equal to the expression to evaluate and there is no default value specified, the function returns `NULL`.

For example:

```
DECODE(3,  1, 'a',
      2, 'b',
      3, 'c',
      4, 'd',
      'x') AS answer_1, -- returns c

DECODE(5,  1, 'a',
      2, 'b',
      3, 'c',
      4, 'd',
      'x') AS answer_2, -- returns x, because 5 is not 1, 2, 3, or 4

DECODE(5,  1, 'a',
        2, 'b',
        3, 'c',
        4, 'd') AS answer_3, -- returns NULL
```

❖ 4.11.2. GREATEST() and LEAST()

The `GREATEST()`/`LEAST()` functions take two or more expressions and return the greatest/least values from the list. They can be used to compare various data types, including numbers, strings, and dates. For example:

```
GREATEST(5, 25, 15) AS high_num, -- returns 25

GREATEST('Harry', 'Hermione', 'Ron') AS high_name, -- returns 'Ron'

LEAST(5, 25, 15) AS low_num, -- returns 5

LEAST('Harry', 'Hermione', 'Ron') AS low_name -- returns 'Harry'

LEAST(TO_DATE('07-04-1776', 'MM-DD-YYYY'),
    TO_DATE('09-03-1783', 'MM-DD-YYYY'))
    AS low_date -- returns 04-JUL-76
```

Conclusion

In this lesson, you have learned the purpose of the DUAL table, to create column aliases, to work with calculated fields, and to use many of Oracle's built-in SQL functions.

LESSON 5
Aggregate Functions

Topics Covered

☑ Aggregate functions.

☑ Grouping aggregate data.

☑ Filtering aggregate data.

☑ `ROLLUP()` and `CUBE()`.

Introduction

In this lesson, you will learn to use aggregate functions to return results based on data in multiple rows.

5.1. Introduction to Aggregate Functions

Aggregate functions are used to calculate results using field values from multiple records. Oracle includes many aggregate functions[2], but there are five that are the most frequently used.

Common Aggregate Functions

Aggregate Function	Description
`COUNT()`	Returns the number of rows containing non-`NULL` values in the specified field.
`SUM()`	Returns the sum of the non-`NULL` values in the specified field.
`AVG()`	Returns the average of the non-`NULL` values in the specified field.
`MAX()`	Returns the maximum of the non-`NULL` values in the specified field.
`MIN()`	Returns the minimum of the non-`NULL` values in the specified field.

2. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Aggregate-Functions.html

Demo 5.1: Aggregate-Functions/Demos/count.sql

```
1.    SELECT COUNT(*) AS num_employees
2.    FROM employees;
3.
4.    SELECT COUNT(*) AS num_sales_managers
5.    FROM employees
6.    WHERE job_id = 'SA_MAN';
7.
8.    SELECT COUNT(manager_id) AS employees_with_bosses
9.    FROM employees;
```

Code Explanation

Using the asterisk (*) within `COUNT(*)` ensures that every row will be counted. Another way to do this is to use the primary key (e.g. `COUNT(employee_id)`) because the primary key cannot be `NULL`. You should find that there are 107 employees, 5 sales managers, and 106 employees without bosses.

Demo 5.2: Aggregate-Functions/Demos/sum.sql


```
1.    SELECT SUM(salary) AS total_salary
2.    FROM employees;
3.
4.    SELECT TO_CHAR(SUM(salary), '$999,999.99') AS total_salary
5.    FROM employees;
```

Code Explanation

If we add up all the employees' salaries, the result is $691,416.00. The first query returns the result as a number and the second query returns the result as a formatted string.

Demo 5.3: Aggregate-Functions/Demos/avg.sql

```
1.    SELECT AVG(salary) AS avg_salary
2.    FROM employees;
3.
4.    SELECT TO_CHAR(AVG(salary), '$999,999.99') AS avg_salary
5.    FROM employees;
```

Code Explanation

The average salary is $6,461.83. The first query returns the result as a number and the second query returns the result as a formatted string.

Demo 5.4: Aggregate-Functions/Demos/min-max.sql

```
1.   SELECT MIN(salary) AS min_salary,
2.     MAX(salary) AS max_salary
3.   FROM employees;
4.
5.   SELECT TO_CHAR(MIN(salary), '$999,999.99') AS min_salary,
6.     TO_CHAR(MAX(salary), '$999,999.99') AS max_salary
7.   FROM employees;
8.
9.   SELECT MIN(hire_date) AS first_hire_date,
10.    MAX(hire_date) AS last_hire_date
11.  FROM employees;
```


Code Explanation

The minimum and maximum salaries are $2,100.00 and $24,000.00.

The first employee was hired on 13-JAN-01 and the most newest employee was hired on 21-APR-08.

Exercise 18: Working with Aggregate Functions

20 to 30 minutes

In this exercise, you will practice working with aggregate functions.

1. Open a new SQL Worksheet in SQL Developer and save it as `aggregates.sql` in `Aggregate-Functions/Exercises`.
2. Write a query that returns the following columns from the `jobs` table:
    A. The sum of all the maximum salaries minus the sum of all the minimum salaries. There are at least two ways you can do this.
    B. The average maximum salary minus the average minimum salary rounded to the penny. There are at least two ways you can do this.
    C. The difference between the highest and lowest **minimum** salaries.
    D. The difference between the highest and lowest **maximum** salaries.
3. Write a query that gets the highest, lowest and average commission percentages from the `employees` table. Round the average to two places after the decimal.
4. Write a query that returns the following columns from the `locations` table:
    A. The total number of locations.
    B. The total number of locations that have a non-`NULL` value for `state_province`.
5. Remember that you can get the number of days employees have been working for the company by subtracting their `hire_date` from `CURRENT_DATE`. And you can get the number of years employees have been working by dividing that result by 365.25:

    ```
    (CURRENT_DATE - hire_date) / 365.25
    ```

    Write a query that gets the total number of years that all employees have worked at the company. For example if the company had three employees who had each worked 10 years, the total years would be 30. Round the result to two decimal places after the decimal point.

Solution: Aggregate-Functions/Solutions/aggregates.sql

```
1.  /* Calculate the following:
2.    - Sum of all the max salaries minus sum of all the min salaries.
3.    - Average max salary minus the average min salary. Round this.
4.    - The difference between the highest and lowest min salaries.
5.    - The difference between the highest and lowest max salaries.
6.  */
7.  SELECT SUM(max_salary) - SUM(min_salary) AS summax_minus_summin_1,
8.    SUM(max_salary - min_salary) AS summax_minus_summin_2,
9.    ROUND(AVG(max_salary) - AVG(min_salary)) AS avgmax_minus_avgmin_1,
10.   ROUND(AVG(max_salary - min_salary)) AS avgmax_minus_avgmin_2,
11.   MAX(min_salary) - MIN(min_salary) AS highmin_minus_lowmin,
12.   MAX(max_salary) - MIN(max_salary) AS highmax_minus_lowmax
13. FROM jobs;
14.
15. -- The highest, lowest and average commission percentages
16. -- Round the average to two places after the decimal
17. SELECT MAX(commission_pct) AS high_commission,
18.   MIN(commission_pct) AS low_commission,
19.   ROUND(AVG(commission_pct),2) AS avg_commission
20. FROM employees;
21.
22. -- How many locations are there? How many have a state_province?
23. SELECT COUNT(*) AS total_locations,
24.   COUNT(STATE_PROVINCE) AS locations_with_state_province
25. FROM locations;
26.
27. -- How many locations have no state_province?
28. SELECT COUNT(*)
29. FROM locations
30. WHERE state_province IS NULL;
31.
32. --Get the total number of years that employees have been working
33. SELECT ROUND(SUM(CURRENT_DATE - hire_date)/365.25, 2) AS person_years
34. FROM employees;
35.
```


5.2. Grouping Data

❖ 5.2.1. GROUP BY

With the `GROUP BY` clause, aggregate functions can be applied to groups of records based on column values. Whenever the `SELECT` clause of your `SELECT` statement includes *non-grouped column values* (values not included in aggregate functions) and aggregate functions, you must include the `GROUP BY` clause. The `GROUP BY` clause must list all of the non-grouped column values used in the `SELECT` clause.

For example, the following code will return the number of employees for each manager:

Demo 5.5: Aggregate-Functions/Demos/group-by.sql

```
1.    SELECT manager_id, COUNT(*) AS num_employees
2.    FROM employees
3.    GROUP BY manager_id
4.    ORDER BY num_employees DESC;
```

Code Explanation

Notice the `ORDER BY` clause comes after the `GROUP BY` clause. You must first group the results before sorting them.

The query will output the following (first 10 rows shown):

	MANAGER_ID	NUM_EMPL...
1	100	14
2	120	8
3	124	8
4	121	8
5	123	8
6	122	8
7	145	6
8	146	6
9	147	6
10	149	6

You can also group by multiple fields as shown below:

Demo 5.6: Aggregate-Functions/Demos/group-by-multiple-fields.sql

```
1.   SELECT manager_id, job_id,
2.     COUNT(*) AS num_employees
3.   FROM employees
4.   GROUP BY manager_id, job_id
5.   ORDER BY num_employees DESC;
```

Code Explanation

The query will output the following (first 10 rows shown):

	MANAGER_ID	JOB_ID	NUM_EMPLOYEES
1	147	SA_REP	6
2	148	SA_REP	6
3	149	SA_REP	6
4	146	SA_REP	6
5	145	SA_REP	6
6	108	FI_ACCOUNT	5
7	114	PU_CLERK	5
8	100	ST_MAN	5
9	100	SA_MAN	5
10	103	IT_PROG	4

Remember that every field in the `SELECT` clause that is not in an aggregate function must also appear in the `GROUP BY` clause.

❖ 5.2.2. HAVING

The `HAVING` clause is used to filter grouped data. Therefore, whenever you want to filter data based on an aggregate function, you do so in the `HAVING` clause. For example, the following code specifies that we only want information on managers that have more than seven employees reporting to them.

Demo 5.7: Aggregate-Functions/Demos/having.sql

```
1.   SELECT manager_id, COUNT(*) AS num_employees
2.   FROM employees
3.   GROUP BY manager_id
4.   HAVING COUNT(*) > 7
5.   ORDER BY num_employees DESC;
```

The query will output the following:

	MANAGER_ID	NUM_EMPLOYEES
1	100	14
2	121	8
3	123	8
4	122	8
5	124	8
6	120	8

❖ 5.2.3. Order of Clauses

1. `SELECT`
2. `FROM`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

❖ 5.2.4. Grouping Rules

- Every non-aggregate column that appears in the `SELECT` clause must also appear in the `GROUP BY` clause.
- You **may not use** aliases in the `HAVING` clause.
- You **may use** aliases in the `ORDER BY` clause.
- When you want to filter out rows from your result set based on aggregate functions, you must use the `HAVING` clause.
- You should declare column aliases for any calculated fields in the `SELECT` clause.
- You may not use aggregate functions in the `WHERE` clause.

Exercise 19: Grouping Results

20 to 30 minutes

1. Open a new SQL Worksheet in SQL Developer and save it as `group-by.sql` in `Aggregate-Functions/Exercises`.
2. Write a query that returns the number of employees by department id. Sort from biggest to smallest departments.
3. Write a query that returns the department ids for all departments that only have one employee.
4. Write a query that returns the most recent hire date by job id. Sort so that most recent hire dates show up first.
5. Write a query that returns the average commission percentage by year hired. Sort by year hired and do not include any years in which no employees hired that year earn a commission (i.e., don't get `NULL` values).

Challenge

Change the last query so that it outputs the commission as a percentage (e.g., `35%` instead of `.35`).

Solution: Aggregate-Functions/Solutions/group-by.sql

```
1.   -- Number of employees by department id
2.   SELECT department_id, COUNT(*) AS num_employees
3.   FROM employees
4.   WHERE department_id IS NOT NULL
5.   GROUP BY department_id
6.   ORDER BY num_employees DESC;
7.
8.   -- Departments that only have one employee
9.   SELECT department_id, COUNT(*) AS num_employees
10.  FROM employees
11.  WHERE department_id IS NOT NULL
12.  GROUP BY department_id
13.  HAVING COUNT(*) = 1;
14.
15.  -- Most recent hire date by job id
16.  SELECT job_id, MAX(hire_date) AS last_hire_date
17.  FROM employees
18.  GROUP BY job_id
19.  ORDER BY last_hire_date DESC;
20.
21.  -- Average commission by year hired
22.  SELECT TO_CHAR(hire_date, 'YYYY') AS year_hired,
23.    ROUND(AVG(commission_pct), 2) AS avg_commission
24.  FROM employees
25.  WHERE commission_pct IS NOT NULL
26.  GROUP BY TO_CHAR(hire_date, 'YYYY')
27.  ORDER BY year_hired;
28.
29.  -- Challenge: Average commission percentage by year hired
30.  SELECT TO_CHAR(hire_date, 'YYYY') AS year_hired,
31.    ROUND(AVG(commission_pct * 100)) || '%' AS avg_commission
32.  FROM employees
33.  WHERE commission_pct IS NOT NULL
34.  GROUP BY TO_CHAR(hire_date, 'YYYY')
35.  ORDER BY year_hired;
```


5.3. Selecting Distinct Records

The `DISTINCT` keyword is used to select distinct combinations of column values. For example, the following example shows how you would find all the distinct departments in the `employees` table:

```
SELECT DISTINCT(department_id) AS department_id
FROM employees
ORDER BY department_id;
```

`DISTINCT` is often used with aggregate functions. The following example shows how `DISTINCT` can be used to find out in how many different departments have employees working in them:

```
SELECT COUNT(DISTINCT(department_id)) AS num_active_departments
FROM employees
ORDER BY department_id;
```

This will return 11. Compare that to the number of departments in the `departments` table:

```
SELECT COUNT(*) AS num_departments
FROM departments;
```

That will return 27.

Open `Aggregate-Functions/Demos/distinct.sql` to run these queries.

5.4. ROLLUP() and CUBE()

The code examples in this section are all in `Aggregate-Functions/Demos/rollup-cube.sql`.

The `ROLLUP()` and `CUBE()` functions provide tallies of grouped data. To understand this, examine the following query:

```
SELECT manager_id, job_id, COUNT(*) AS num_employees
FROM employees
WHERE manager_id IS NOT NULL
GROUP BY manager_id, job_id
ORDER BY manager_id, job_id;
```

The first 6 and the last 6 results of this query are shown below:

	MANAGER_ID	JOB_ID	NUM_EMPLOYEES
1	100	AD_VP	2
2	100	MK_MAN	1
3	100	PU_MAN	1
4	100	SA_MAN	5
5	100	ST_MAN	5
6	101	AC_MGR	1
26	146	SA_REP	6
27	147	SA_REP	6
28	148	SA_REP	6
29	149	SA_REP	6
30	201	MK_REP	1
31	205	AC_ACCOUNT	1

The results show that the employee with the id of 100 has the following reportees:

1. 2 AD_VPs
2. 1 MK_MAN
3. 1 PU_MAN
4. 5 SA_MANs
5. 5 ST_MANs

But it doesn't show how many reportees they have in total.

❖ 5.4.1. ROLLUP()

Using the `ROLLUP()` function, we can get that too:

```
SELECT manager_id, job_id, COUNT(*) AS num_employees
FROM employees
WHERE manager_id IS NOT NULL
GROUP BY ROLLUP(manager_id, job_id)
ORDER BY manager_id, job_id;
```

The first 6 and the last 6 results of this query are shown below:

	MANAGER_ID	JOB_ID	NUM_EMPLOYEES
1	100	AD_VP	2
2	100	MK_MAN	1
3	100	PU_MAN	1
4	100	SA_MAN	5
5	100	ST_MAN	5
6	100	(null)	14
45	149	(null)	6
46	201	MK_REP	1
47	201	(null)	1
48	205	AC_ACCOUNT	1
49	205	(null)	1
50	(null)	(null)	106

Notice how this gives us a tally of the number of employees who report to each manager and a tally of the number of employees who report to any manager.

Note that you can use the `COALESCE()` function to replace the `NULL` values with meaningful values:

```
SELECT COALESCE(TO_CHAR(manager_id), '** All Managers') AS manager_id,
  COALESCE(job_id, '** All Jobs') AS job_id,
  COUNT(*) AS num_employees
FROM employees
WHERE manager_id IS NOT NULL
GROUP BY ROLLUP(manager_id, job_id)
ORDER BY manager_id, job_id;
```

This will return the following (first 10 rows shown):

	MANAGER_ID	JOB_ID	NUM_EMPLOYEES
1	** All Managers	** All Jobs	106
2	100	** All Jobs	14
3	100	AD_VP	2
4	100	MK_MAN	1
5	100	PU_MAN	1
6	100	SA_MAN	5
7	100	ST_MAN	5
8	101	** All Jobs	5
9	101	AC_MGR	1
10	101	AD_ASST	1

Prefixing "All Managers" and "All Jobs" with asterisks makes these values stand out and assures that, when sorted, they will show up before the other values.

❖ 5.4.2. CUBE()

`CUBE()` takes it one step further: it also gives us a tally of the number of employees with each job id:


```
SELECT COALESCE(TO_CHAR(manager_id), '** All Managers') AS manager_id,
  COALESCE(job_id, '** All Jobs') AS job_id,
  COUNT(*) AS num_employees
FROM employees
WHERE manager_id IS NOT NULL
GROUP BY CUBE(manager_id, job_id)
ORDER BY manager_id, job_id;
```

The last 10 results of this query are shown below:

59	(null)	MK_REP	1
60	(null)	PR_REP	1
61	(null)	PU_CLERK	5
62	(null)	PU_MAN	1
63	(null)	SA_MAN	5
64	(null)	SA_REP	30
65	(null)	SH_CLERK	20
66	(null)	ST_CLERK	20
67	(null)	ST_MAN	5
68	(null)	(null)	106

You should run this query in SQL Developer to see all the results. It gives everything the previous query with `ROLLUP()` gave us, but after rolling up (i.e., tallying) all the employees for each manager id and job id, it give tallies of the number of employees with each job id, regardless of who their manager is.

Conclusion

In this lesson, you have learned to work with Oracle's aggregate functions.

LESSON 6
Joins

Topics Covered

☑ Inner joins.

☑ Outer joins.

Introduction

In this lesson, you will learn to write queries that return data from multiple tables using SQL joins.

6.1. Inner Joins

Up until this point, we have only written queries that select data from a single table, but often we need to get data from multiple tables to create useful reports. Consider, for example, the following query, which selects all four columns from the `departments` table:

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

Here are the first 10 results:

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500
6	60	IT	103	1400
7	70	Public Relations	204	2700
8	80	Sales	145	2500
9	90	Executive	100	1700
10	100	Finance	108	1700

The only column in this report that has meaningful values to the average person is the department name. The other fields are ids referencing other tables. Joins allow us to use those fields to join the departments table with other tables that have more meaningful data. The basic syntax for a join is:

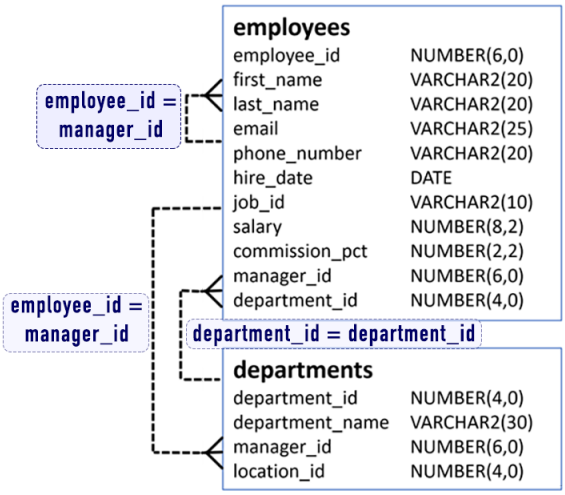
```
SELECT table_name_1.column_name, table_name_1.column_name,
  table_name_2.column_name, table_name_2.column_name, ...
FROM table_name_1
  JOIN table_name_2 ON
    table_name_1.pk_field = table_name_2.fk_field;
```

`pk_field` represents the primary key field in table 1 (e.g., `department_id`) and `fk_field` represents the foreign key field in table 2. It is through this primary key - foreign key relationship that the table relate to each other. Although it is possible to join tables on non-key fields, it is uncommon.

Because it gets unwieldy to repeat the table names, it is common practice to use table aliases like this:

```
SELECT tn1.column_name, tn1.column_name,
        tn2.column_name, tn2.column_name, ...
FROM table_name_1 tn1
  JOIN table_name_2 tn2 ON tn1.pk_field = tn2.fk_field;
```


Consider the following relationship diagram between the `employees` and `departments` tables:



Note the following:

1. The `manager_id` field in the `departments` table is a foreign key referencing the `employee_id` field in the `employees` table. This tells us who the manager is for each department.

2. The `department_id` field in the `employees` table is a foreign key referencing the `department_id` field in the `departments` table. This tells us which department each employee is in.
3. The `manager_id` field in the `employees` table is a foreign key referencing the `employee_id` field in the *same* `employees` table. This tells us the manager of each employee.

The following queries both return the department name and manager for each department with a manager. The first does not use table aliases and the second does:

```
-- No table aliases
SELECT departments.department_name,
  employees.first_name || ' ' || employees.last_name AS manager
FROM departments
  JOIN employees ON employees.employee_id = departments.manager_id;

-- The same query using table aliases
SELECT d.department_name,
  e.first_name || ' ' || e.last_name AS manager
FROM departments d
  JOIN employees e ON e.employee_id = d.manager_id;
```


The join shown above is called an *inner join*. With inner joins, records are only output when a relationship is found. In other words, employees who are not managers of any department are not returned by the query above. Likewise, departments that have no assigned manager are also not returned. This query can be made explicit by adding the keyword `INNER`:

```
SELECT d.department_name,
e.first_name || ' ' || e.last_name AS manager
FROM departments d
  INNER JOIN employees e ON e.employee_id = d.manager_id;
```

But as joins are inner joins by default, the `INNER` keyword is usually left out.

After an exercise on inner joins, you will learn about outer joins, which return data from one or both of the joined tables even if no matching data is found in the joining table.

Exercise 20: Inner Joins

30 to 45 minutes

In this exercise, you will practice writing inner joins. You will likely find it useful to reference the PDF in `Relational-Database-Basics/hr-entity-diagram.pdf`.

1. Open a new SQL Worksheet in SQL Developer and save it as `inner-joins.sql` in `Joins/Exercises`.
2. Create a report that returns employees by department, sorted first by the name of the department and then by the employee's last name.

	DEPARTMENT_NAME	EMPLOYEE
1	Accounting	William Gietz
2	Accounting	Shelley Higgins
3	Administration	Jennifer Whalen
4	Executive	Lex De Haa
5	Executive	Steven King
6	Executive	Neena Kochhar
7	Finance	John Chen
8	Finance	Daniel Faviet
9	Finance	Nancy Greenberg
10	Finance	Luis Popp

3. Create a report that shows the city, state/province, and country name of each location. Sort by country name. The first 10 rows should look like this:

	CITY	STATE_PROVINCE	COUNTRY_NAME
1	Sydney	New South Wales	Australia
2	Sao Paulo	Sao Paulo	Brazil
3	Toronto	Ontario	Canada
4	Whitehorse	Yukon	Canada
5	Beijing	(null)	China
6	Munich	Bavaria	Germany
7	Bombay	Maharashtra	India
8	Roma	(null)	Italy
9	Venice	(null)	Italy
10	Tokyo	Tokyo Prefecture	Japan

4. You are not limited to joining on just two tables. Expand the report above to include the region name:

	CITY	STATE_PROVINCE	COUNTRY_NAME	REGION_NAME
1	Sydney	New South Wales	Australia	Asia
2	Sao Paulo	Sao Paulo	Brazil	Americas
3	Toronto	Ontario	Canada	Americas
4	Whitehorse	Yukon	Canada	Americas
5	Beijing	(null)	China	Asia
6	Munich	Bavaria	Germany	Europe
7	Bombay	Maharashtra	India	Asia
8	Roma	(null)	Italy	Europe
9	Venice	(null)	Italy	Europe
10	Hiroshima	(null)	Japan	Asia

5. Create a report that shows the manager's full name and job title for each department, and also shows the number of employees in that department. The full report is shown below:

	MANAGER	JOB_TITLE	DEPARTMENT_NAME	NUM_EMPLOYEES
1	Adam Fripp	Stock Manager	Shipping	45
2	John Russell	Sales Manager	Sales	34
3	Nancy Greenberg	Finance Manager	Finance	6
4	Den Raphaely	Purchasing Manager	Purchasing	6
5	Alexander Hunold	Programmer	IT	5
6	Steven King	President	Executive	3
7	Michael Hartstein	Marketing Manager	Marketing	2
8	Shelley Higgins	Accounting Manager	Accounting	2
9	Hermann Baer	Public Relations Representative	Public Relations	1
10	Jennifer Whalen	Administration Assistant	Administration	1
11	Susan Mavris	Human Resources Representative	Human Resources	1

- **Hint** (*skip this if you don't want a hint*): Start with the `employees` table to get the manager's full name, then join the `jobs` table to get the manager's job title, then join the `departments` table to get the department the manager manages, and finally join the `employees` table to the departments table to get the number of employees in that department.

Challenge

1. The `employees` table has a self-referencing primary key - foreign key relationship. The `manager_id` field references the `employee_id` field. Write a query that returns employees and their managers sorted first by the managers' last and first names and then by the employees' last and first names, like this (first 10 results shown):

	MANAGER	EMPLOYEE
1	Gerald Cambrault	Elizabeth Bates
2	Gerald Cambrault	Harrison Bloom
3	Gerald Cambrault	Tayler Fox
4	Gerald Cambrault	Sundita Kumar
5	Gerald Cambrault	Lisa Ozer
6	Gerald Cambrault	William Smith
7	Lex De Haan	Alexander Hunold
8	Alberto Errazuriz	Sundar Ande
9	Alberto Errazuriz	Amit Banda
10	Alberto Errazuriz	Danielle Greene

2. Create a report that shows the number of direct reports by manager. Sort by the number of direct reports in descending order. The first 10 rows should look like this:

	MANAGER	NUM_REPORTS
1	Steven King	14
2	Shanta Vollman	8
3	Kevin Mourgos	8
4	Matthew Weiss	8
5	Adam Fripp	8
6	Payam Kaufling	8
7	Alberto Errazuriz	6
8	John Russell	6
9	Karen Partners	6
10	Eleni Zlotkey	6

Solution: Joins/Solutions/inner-joins.sql

```
1.  -- Employees by department
2.  SELECT d.department_name, e.first_name || ' ' || e.last_name as Employee
3.  FROM employees e
4.    JOIN departments d ON e.department_id = d.department_id
5.  ORDER BY d.department_name, e.last_name;
6.
7.  -- Locations with country name
8.  SELECT l.city, l.state_province,
9.    c.country_name
10. FROM locations l
11.   JOIN countries c ON c.country_id = l.country_id
12. ORDER BY country_name;
13.
14. -- Locations with country and region names
15. SELECT l.city, l.state_province,
16.   c.country_name,
17.   r.region_name
18. FROM locations l
19.   JOIN countries c ON c.country_id = l.country_id
20.   JOIN regions r ON r.region_id = c.region_id
21. ORDER BY country_name;
22.
23. -- Departments, their managers and number of employees
24. SELECT m.first_name || ' ' || m.last_name AS manager,
25.   j.job_title,
26.   d.department_name,
27.   COUNT(e.employee_id) AS num_employees
28. FROM employees m -- for managers
29.   JOIN jobs j ON j.job_id = m.job_id
30.   JOIN departments d ON m.employee_id = d.manager_id
31.   JOIN employees e ON d.department_id = e.department_id
32. GROUP BY m.first_name, m.last_name, j.job_title, d.department_name
33. ORDER BY num_employees DESC;
34.
35. /* CHALLENGES */
36. -- Employees and their managers
37. SELECT m.first_name || ' ' || m.last_name AS manager,
38.   e.first_name || ' ' || e.last_name AS employee
39. FROM employees m
40.   JOIN employees e ON m.employee_id = e.manager_id
41. ORDER BY m.last_name, m.first_name, e.last_name, e.first_name;
42.
43. -- Num direct reports by manager
44. SELECT m.first_name || ' ' || m.last_name AS manager,
```

```
45.     COUNT(e.employee_id) AS num_reports
46.   FROM employees m
47.     JOIN employees e ON m.employee_id = e.manager_id
48.   GROUP BY m.first_name, m.last_name
49.   ORDER BY num_reports DESC;
```

6.2. Outer Joins

So far, all the joins we have worked with are inner joins, meaning that rows are only returned that have matches in both tables. For example, when doing an inner join between the `employees` table and the `departments` table on the department manager, only employees that manage departments and departments that have managers will be returned. Here's the query again:

```
SELECT d.department_name,
  e.first_name || ' ' || e.last_name AS manager
FROM departments d
  JOIN employees e ON e.employee_id = d.manager_id;
```


❖ 6.2.1. Left Joins

A `LEFT JOIN` (also called a `LEFT OUTER JOIN`) returns all the records from the first table even if there are no matches in the second table.

```
SELECT t1.column_name, t2.column_name
FROM table_name_1 t1
  LEFT JOIN table_name_2 t2 ON t1.column_name = t2.column_name
```

Tables listed earlier in the `FROM` clause are on the left, so all rows in `table_name_1` will be returned even if they do not have matches in `table_name_2`.

So, just adding the keyword `LEFT` to our department-manager query, we get:

```
SELECT d.department_name,
  e.first_name || ' ' || e.last_name AS manager
FROM departments d
  LEFT JOIN employees e ON e.employee_id = d.manager_id;
```

Because `departments` is listed before `employees` in the `FROM` clause, it's considered to be on the left, so this query will return all departments, even those without managers, but it will still only return employees who manage departments. Here are rows 6 through 15:

	DEPARTMENT_NAME	MANAGER
6	Sales	John Russell
7	Administration	Jennifer Whalen
8	Marketing	Michael Hartstein
9	Human Resources	Susan Mavris
10	Public Relations	Hermann Baer
11	Accounting	Shelley Higgins
12	Treasury	
13	Corporate Tax	
14	Control And Credit	
15	Shareholder Services	

❖ 6.2.2. Right Joins

A `RIGHT JOIN` (also called a `RIGHT OUTER JOIN`) returns all the records from the second table even if there are no matches in the first table.


```
SELECT t1.column_name, t2.column_name
FROM table_name_1 t1
  RIGHT JOIN table_name_2 t2 ON t1.column_name = t2.column_name
```

Tables listed earlier in the `FROM` clause are on the left, so all rows in `table_name_2` will be returned even if they do not have matches in `table_name_1`.

So, just adding the keyword `RIGHT` to our department-manager query, we get:

```
SELECT d.department_name,
  e.first_name || ' ' || e.last_name AS manager
FROM departments d
  RIGHT JOIN employees e ON e.employee_id = d.manager_id;
```

Because `employees` is listed after `departments` in the `FROM` clause, it's considered to be on the right, so this query will return all employees, even those who do not manager departments, but it will only return departments that have managers. Here are rows 6 through 15:

	DEPARTMENT_NAME	MANAGER
6	IT	Alexander Hunold
7	Public Relations	Hermann Baer
8	Sales	John Russell
9	Executive	Steven King
10	Finance	Nancy Greenberg
11	Accounting	Shelley Higgins
12	(null)	Diana Lorentz
13	(null)	Sundita Kumar
14	(null)	Kevin Mourgos
15	(null)	Sarath Sewall

❖ 6.2.3. Full Outer Joins

A `FULL JOIN` (also called a `FULL OUTER JOIN`) returns all the records from each table even if there are no matches in the joined table.

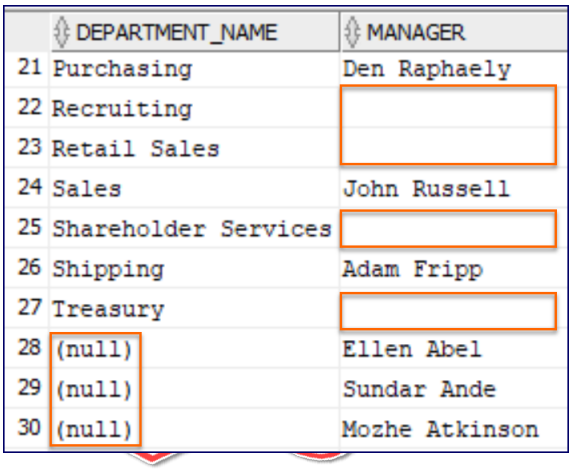

```
SELECT t1.column_name, t2.column_name
FROM table_name_1 t1
  FULL JOIN table_name_2 t2 ON t1.column_name = t2.column_name
```

All rows in `table_name_1` and `table_name_2` will be returned.

So, just adding the keyword `FULL` to our department-manager query, we get:

```
SELECT d.department_name,
  e.first_name || ' ' || e.last_name AS manager
FROM departments d
  FULL JOIN employees e ON e.employee_id = d.manager_id;
```

This query will return all employees, even those who do not manager departments. It will also return all departments, even those without managers. Here are rows 21 through 30:



	DEPARTMENT_NAME	MANAGER
21	Purchasing	Den Raphaely
22	Recruiting	
23	Retail Sales	
24	Sales	John Russell
25	Shareholder Services	
26	Shipping	Adam Fripp
27	Treasury	
28	(null)	Ellen Abel
29	(null)	Sundar Ande
30	(null)	Mozhe Atkinson

Notice that there are `NULL` values for departments that don't have managers, but there are just blanks for the employees who dont manage departments. Those blanks are actually single spaces. It is because the field is populated with `e.first_name || ' ' || e.last_name`, which for departments with no manager, evaluates to `NULL || ' ' || NULL`. When used in concatenation, `NULL` gets implicitly converted to an empty string, so this further evaluates to `'' || ' ' || ''` or just a single space: `' '`.

Exercise 21: Outer Joins

30 to 45 minutes

In this exercise, you will practice writing outer joins. You will likely find it useful to reference the PDF in `Relational-Database-Basics/hr-entity-diagram.pdf`.

1. Open a new SQL Worksheet in SQL Developer and save it as `outer-joins.sql` in `Joins/Exercises`.
2. Create a report that shows the number of employees by department. Include all departments, even those with no employees. The first 10 rows should look like this:

	DEPARTMENT_NAME	NUM_EMPLOYEES
1	Accounting	2
2	Administration	1
3	Benefits	0
4	Construction	0
5	Contracting	0
6	Control And Credit	0
7	Corporate Tax	0
8	Executive	3
9	Finance	6
10	Government Sales	0

3. Create a report that shows the city, state/province, and country name of each location. Include all countries even if there are no matching locations. Sort by country name. The first 10 rows should look like this:

	CITY	STATE_PROVINCE	COUNTRY_NAME
1	(null)	(null)	Argentina
2	Sydney	New South Wales	Australia
3	(null)	(null)	Belgium
4	Sao Paulo	Sao Paulo	Brazil
5	Toronto	Ontario	Canada
6	Whitehorse	Yukon	Canada
7	Beijing	(null)	China
8	(null)	(null)	Denmark
9	(null)	(null)	Egypt
10	(null)	(null)	France

4. Create the same report as above, but this time replace the `NULL` values in the `city` and `state_province` columns with "N/A" for Not Applicable. You will need to use one fo the `NULL`-related functions. The first 10 rows should look like this:

	CITY	STATE_PROVINCE	COUNTRY_NAME
1	N/A	N/A	Argentina
2	Sydney	New South Wales	Australia
3	N/A	N/A	Belgium
4	Sao Paulo	Sao Paulo	Brazil
5	Toronto	Ontario	Canada
6	Whitehorse	Yukon	Canada
7	Beijing	N/A	China
8	N/A	N/A	Denmark
9	N/A	N/A	Egypt
10	N/A	N/A	France

Challenge

Create a report that shows department managers, their job titles, the name of the department they manage, and the number of employees in the department. Show all departments that have five or fewer employees, even if they have no manager and no employees:

	MANAGER	JOB_TITLE	DEPARTMENT_NAME	NUM_EMPLOYEES
1	Alexander Hunold	Programmer	IT	5
2	Steven King	President	Executive	3
3	Michael Hartstein	Marketing Manager	Marketing	2
4	Shelley Higgins	Accounting Manager	Accounting	2
5	Hermann Baer	Public Relations Representative	Public Relations	1
6	Susan Mavris	Human Resources Representative	Human Resources	1
7	Jennifer Whalen	Administration Assistant	Administration	1
8		(null)	Government Sales	0
9		(null)	Construction	0
10		(null)	NOC	0

Hint: You do not need the `manager_id` field from the `employees` table for this query. That field gives you the manager for a specific employee. But you need the manager for a department. Study the **Primary Key - Foreign Key Relationships** in the `hr-entity-diagram.pdf`. Pay particular attention to the primary key referenced by the `manager_id` field in the `departments` table.

Solution: Joins/Solutions/outer-joins.sql

```
1.  -- Employees by department
2.  SELECT d.department_name,
3.    COUNT(e.employee_id) AS num_employees
4.  FROM departments d
5.    LEFT JOIN employees e ON d.department_id = e.department_id
6.  GROUP BY d.department_name
7.  ORDER BY d.department_name;
8.
9.  -- Locations with country name
10. SELECT l.city, l.state_province,
11.   c.country_name
12. FROM locations l
13.   RIGHT JOIN countries c ON c.country_id = l.country_id
14. ORDER BY country_name;
15.
16. -- Locations with country name using COLAESCE
17. SELECT COALESCE(l.city, 'N/A') AS city,
18.   COALESCE(l.state_province, 'N/A') AS state_province,
19.   c.country_name
20. FROM locations l
21.   RIGHT JOIN countries c ON c.country_id = l.country_id
22. ORDER BY country_name;
23.
24. -- CHALLENGE
25. -- Managers, their job titles, dept name, and num employees in dept
26. SELECT m.first_name || ' '  || m.last_name AS manager,
27.   j.job_title,
28.   d.department_name,
29.   COUNT(e.employee_id) AS num_employees
30. FROM employees m -- for managers
31.   JOIN jobs j ON j.job_id = m.job_id
32.   RIGHT JOIN departments d ON m.employee_id = d.manager_id
33.   LEFT JOIN employees e ON d.department_id = e.department_id
34. GROUP BY m.first_name, m.last_name, j.job_title, d.department_name
35. HAVING COUNT(e.employee_id) <= 5
36. ORDER BY num_employees DESC;
```


Conclusion

In this lesson, you have learned to write inner and outer joins.

LESSON 7
Subqueries

Topics Covered

☑ Subqueries.

Introduction

Subqueries are queries embedded in queries. They are used to retrieve data from one table based on data in another table.

7.1. Subquery Basics

Subquery Demos

All of the demo queries in this section are in the `Subqueries/Demos/subqueries.sql` file.

Please open that file in SQL Developer to follow along with the demos.

Subqueries generally are used when tables have some kind of relationship. For example, in the HR database, the `employees` table has a `job_id` field, which references a job in the `jobs` table. Retrieving the `job_id` for a specific employee is pretty straightforward:

```
SELECT job_id
FROM employees
WHERE first_name = 'Steven' AND last_name = 'King';
```

This will return `AD_PRES`, which might not be meaningful to the people reading the report. Knowing this job id, we can get the job title from the `jobs` table:

```
SELECT job_title
FROM jobs
WHERE job_id = 'AD_PRES';
```

This will return `President`.

But having to run one query to get one piece of data and then another query using that data to get the data you really want is a bit of a burden.

The next query combines both queries into a single query:

```
SELECT job_title
FROM jobs
WHERE job_id = (
          SELECT job_id
          FROM employees
          WHERE first_name = 'Steven' AND last_name = 'King'
        );
```


The inner query gets resolved first and the result is used by the outer query.

If the condition in the `WHERE` clause is checking for a single value (as it does with operators such as `=`, `<>`, and `LIKE`), the inner query must return one and only one result. For example, the following query would cause an error because the inner query (`SELECT job_id FROM employees WHERE last_name = 'King'`) returns more than one result:

```
SELECT job_title
FROM jobs
WHERE job_id = (
          SELECT job_id
          FROM employees
          WHERE last_name = 'King'
        );
```

If you're unsure how many results the inner query will return, then it's safer to use `IN`:

```
SELECT job_title
FROM jobs
WHERE job_id IN (
          SELECT job_id
          FROM employees
          WHERE last_name = 'King'
        );
```

Note that the inner query can only select one column, but that the outer query can select as many as you like:


```
SELECT job_title, job_id, min_salary, max_salary
FROM jobs
WHERE job_id IN (
          SELECT job_id
          FROM employees
          WHERE last_name = 'King'
        );
```

Exercise 22: Subqueries

20 to 30 minutes

In this exercise, you will practice writing subqueries.

1. Open a new SQL Worksheet in SQL Developer and save it as `subqueries.sql` in `Subqueries/Exercises`.
2. Create a report that shows the first and last names of all employees who report to Gerald Cambrault.
3. Create a report that shows all the departments that are not in the United States of America.
    A. First write this knowing that the country id for the United States of America is `US`.
    B. Next write it assuming you do not know the country id for the United States of America. You will need to nest two levels deep.

Solution: Subqueries/Solutions/subqueries.sql

```
1.    -- Names of all employees who report to Gerald Cambrault
2.    SELECT first_name, last_name
3.    FROM employees
4.    WHERE manager_id = (SELECT employee_id
5.                        FROM employees
6.                        WHERE last_name = 'Cambrault'
7.                          AND first_name = 'Gerald');
8.
9.    -- Departments not in the United States (knowing country_id)
10.   SELECT department_name
11.   FROM departments
12.   WHERE location_id NOT IN
13.         (
14.           SELECT location_id
15.           FROM locations
16.           WHERE country_id = 'US'
17.         );
18.
19.   -- Departments not in the United States (w/o knowing country_id)
20.   SELECT department_name
21.   FROM departments
22.   WHERE location_id NOT IN
23.         (
24.           SELECT location_id
25.           FROM locations
26.           WHERE country_id =
27.               (
28.                 SELECT country_id
29.                 FROM countries
30.                 WHERE country_name = 'United States of America'
31.               )
32.         );
```


7.2. Subqueries in the SELECT Clause

Subquery-in-SELECT Demos

All of the demo queries in this section are in the `Subqueries/Demos/subqueries-in-select.sql` file.

Please open that file in SQL Developer to follow along with the demos.

Let's say your manager has asked for a report that returns the following for each employee:

1. First name
2. Last name
3. Job ID
4. Salary
5. Average salary of all employees with the same job id

The first 10 rows of the results are shown below:

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY	AVG_SALARY_JOB
1	Neena	Kochhar	AD_VP	17000	17000
2	Lex	De Haan	AD_VP	17000	17000
3	Daniel	Faviet	FI_ACCOUNT	9000	7920
4	John	Chen	FI_ACCOUNT	8200	7920
5	Ismael	Sciarra	FI_ACCOUNT	7700	7920
6	Jose Manuel	Urman	FI_ACCOUNT	7800	7920
7	Luis	Popp	FI_ACCOUNT	6900	7920
8	Alexander	Khoo	PU_CLERK	3100	2780
9	Shelli	Baida	PU_CLERK	2900	2780
10	Sigal	Tobias	PU_CLERK	2800	2780

The first four fields are easy enough:

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

But getting the average salary of all employees in the same row is tricky. You cannot do it with a `GROUP BY` clause, because you would have to group by all of the other columns in the `SELECT` clause.

The solution is to use a subquery in the `SELECT` clause like this. First, let's get the average salary of all employees:

```
SELECT first_name, last_name, job_id, salary,
  (
      SELECT ROUND(AVG(salary))
      FROM employees
    ) AS avg_salary
FROM employees;
```

Notice the subquery is in parentheses and then gets an alias just like we have seen with other calculated columns.

The next step is to add a WHERE clause to the subquery, but it's a little tricky. We need to compare this employee's salary with the average salary of all employees with the same `job_id`. The easiest way to do that is to give aliases to the two `employees` tables: the one in the main query and the one in the subquery and then compare the `job_id`s from the different tables:


```
SELECT first_name, last_name, job_id, salary,
  (
    SELECT ROUND(AVG(salary))
    FROM employees e2
    WHERE e1.job_id=e2.job_id
  ) AS avg_salary_job
FROM employees e1;
```

The `WHERE` clause of the subquery tells the subquery to get only the employees with a `job_id` that matches the `job_id` of the current employee being returned by the main query.

❖ 7.2.1. Combining SELECT and WHERE Subqueries

Now let's limit our query to only show employees who make more than their colleagues with the same job:

```
SELECT first_name, last_name, job_id, salary,
  (
    SELECT ROUND(AVG(salary))
    FROM employees e2
    WHERE e1.job_id=e2.job_id
  ) AS avg_salary_job
FROM employees e1
WHERE salary > (
      SELECT ROUND(AVG(salary))
      FROM employees e2
      WHERE e1.job_id=e2.job_id
    );
```

Finally, let's show the same employees as above and how much more they earn than their colleague with the same job:


```
SELECT first_name, last_name, job_id, salary,
  (
    SELECT ROUND(AVG(salary))
    FROM employees e2
    WHERE e1.job_id=e2.job_id
  ) AS avg_salary_job,
  salary - (
    SELECT ROUND(AVG(salary))
    FROM employees e2
    WHERE e1.job_id=e2.job_id
  ) AS earning_diff
FROM employees e1
WHERE salary > (
    SELECT ROUND(AVG(salary))
    FROM employees e2
    WHERE e1.job_id=e2.job_id
  )
ORDER BY earning_diff DESC;
```

Exercise 23: Subqueries in SELECTs

10 to 20 minutes

Open a new SQL Worksheet in SQL Developer and save it as `subqueries-in-select.sql` in `Subqueries/Exercises`.

Write a report that shows the first and last names and the hire dates of all employees who share a hire date with one or more other employees and show how many total employees were hired on their hire date. The full report will look like this:

	FIRST_NAME	LAST_NAME	HIRE_DATE	HIRED_SAME_DAY
1	Shelley	Higgins	07-JUN-02	4
2	Hermann	Baer	07-JUN-02	4
3	Susan	Mavris	07-JUN-02	4
4	William	Gietz	07-JUN-02	4
5	Amit	Banda	21-APR-08	2
6	Tayler	Fox	24-JAN-06	2
7	Sundita	Kumar	21-APR-08	2
8	Winston	Taylor	24-JAN-06	2
9	Martha	Sullivan	21-JUN-07	2
10	Anthony	Cabrio	07-FEB-07	2
11	Donald	OConnell	21-JUN-07	2
12	Lindsey	Smith	10-MAR-05	2
13	Peter	Hall	20-AUG-05	2
14	Alberto	Errazuriz	10-MAR-05	2
15	Diana	Lorentz	07-FEB-07	2
16	Laura	Bissot	20-AUG-05	2

Solution: Subqueries/Solutions/subqueries-in-select.sql

```
1.    -- How many employees were hired on the same day as
2.    -- each employee?
3.    SELECT first_name, last_name, hire_date,
4.      (
5.        SELECT count(employee_id)
6.        FROM employees e2
7.        WHERE e2.hire_date = e1.hire_date
8.      ) AS hired_same_day
9.    FROM employees e1
10.   WHERE (SELECT count(employee_id)
11.       FROM employees e2
12.       WHERE e2.hire_date = e1.hire_date) > 1
13.   ORDER BY hired_same_day DESC;
```


Conclusion

In this lesson, you have learned to write subqueries in Oracle.

LESSON 8
Set Operators

Topics Covered

☑ `UNION` and `UNION ALL`.

☑ `INTERSECT`.

☑ `MINUS`.

Introduction

Set operators combine the results of two separate queries into a single result set. In this lesson, we will look at `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`.

8.1. Set Operators

Oracle SQL's four set operators are:

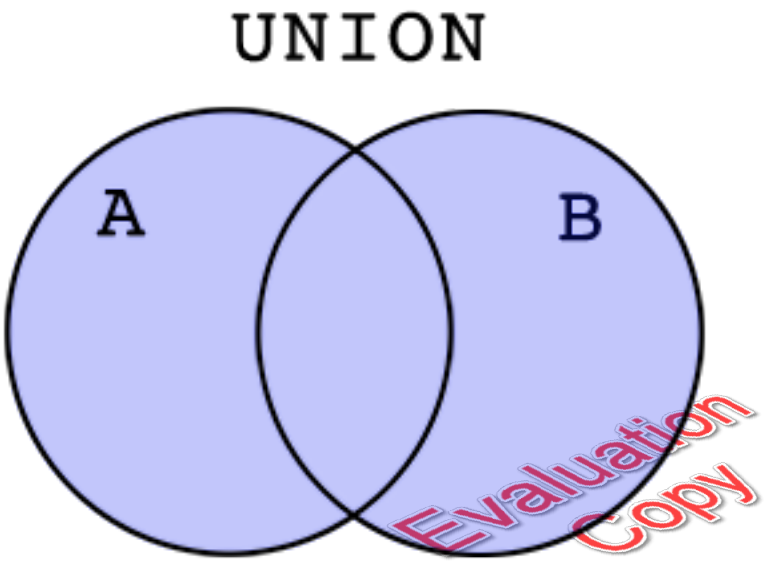
1. `UNION` - returns the combined results of two queries, removing duplicates.
2. `UNION ALL` - returns the combined results of two queries, without removing duplicates.
3. `INTERSECT` - returns only the results that show up in both queries.
4. `MINUS` - returns the results in the first query that do not show up in the second query.

❖ 8.1.1. Rules for Set Operations

1. Each query must return the same number of columns.
2. The columns must be in the same order.
3. Column data types must be compatible.
4. There must be only one `ORDER BY` clause, which must come at the very end.

8.2. UNION

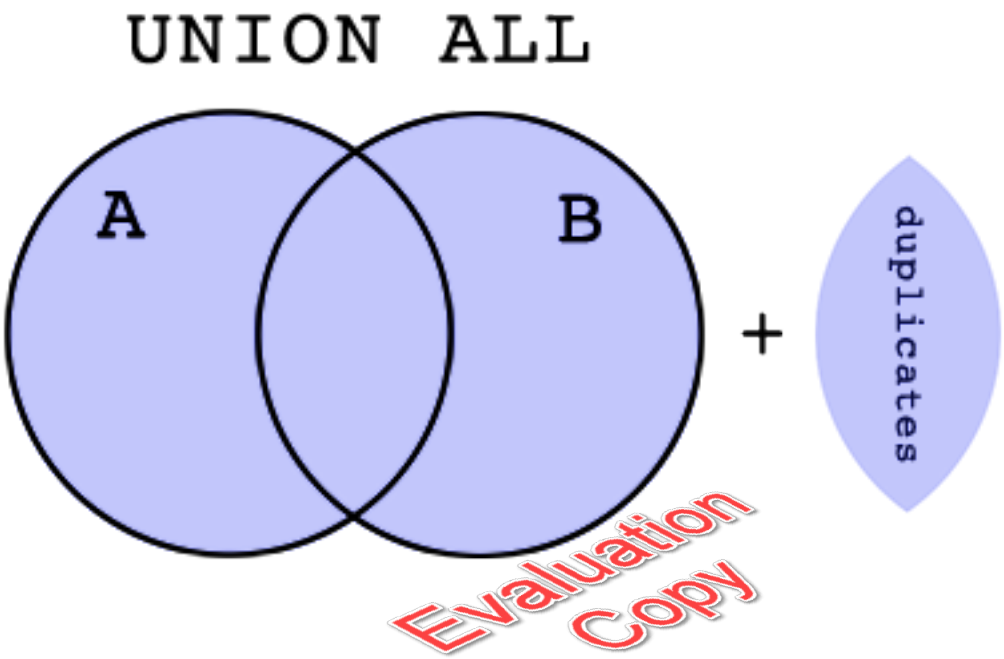
`UNION` is used to get a combination of two result sets with all duplicates removed. It is represented by the Venn diagram below:




```
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
UNION
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
ORDER BY column_name_1
```

8.3. UNION ALL

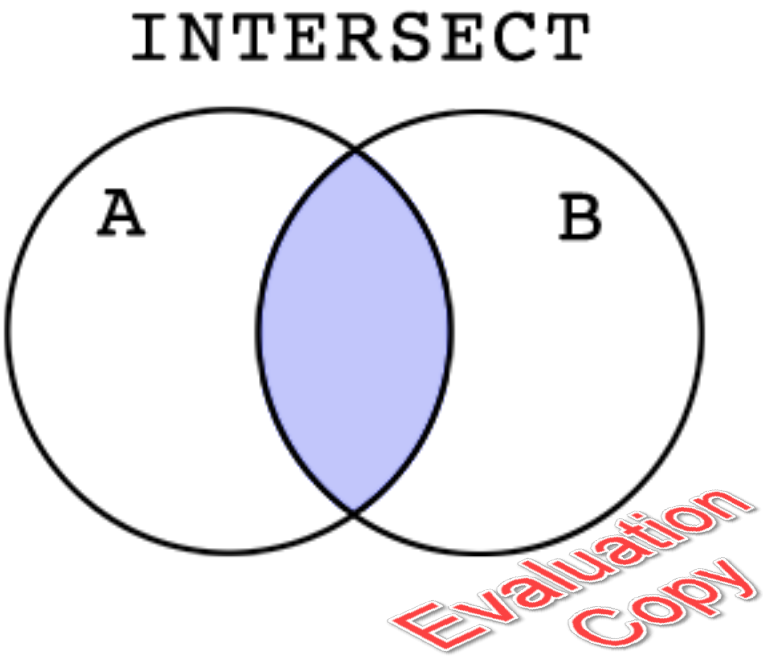
`UNION ALL` is similar to `UNION` except that it returns duplicates. It is represented by the pseudo-Venn diagram below:




```
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
UNION ALL
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
ORDER BY column_name_1
```

8.4. INTERSECT

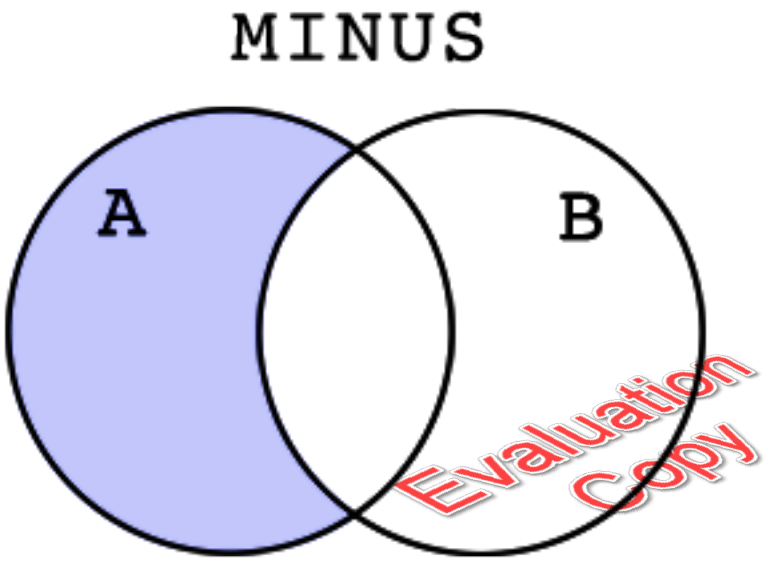
`INTERSECT` is used to get the matching results from two separate queries. It is represented by the Venn diagram below:




```
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
INTERSECT
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
ORDER BY column_name_1
```

8.5. MINUS

`MINUS` is used to get the results from the first query that do not show up in the second query. It is represented by the Venn diagram below:



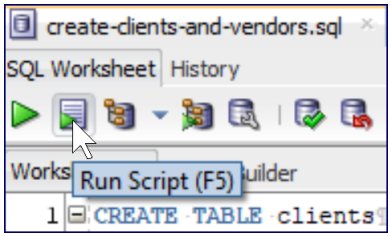

```
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
MINUS
SELECT column_name_1, column_name_2, column_name_3
FROM table_name_1
ORDER BY column_name_1
```

Exercise 24: Working with Set Operators

20 to 30 minutes

In this exercise, you will practice using set operators. Oracle's example HR database does not include good tables for practicing set operations, so we're going to add a couple of new tables.

1. Open `Set-Operators/Exercises/create-clients-and-vendors.sql` in SQL Developer. Press the Run Script icon to run the entire script:



   This will run the entire script, creating and populating new `clients` and `vendors` tables, each with the following fields:

   A. A primary key
   B. `first_name`
   C. `last_name`
   D. `email`
   E. `phone_number`

   Some of the clients are also vendors.

2. Open `Set-Operators/Exercises/set-operators.sql` in SQL Developer.
3. Create a report that returns the first and last names, emails, and phone numbers of all clients and vendors. Do not include duplicates. **This has been done for you.** Sort by last name then first name.
4. Create a report that returns the first and last names, emails, and phone numbers of all clients and vendors. Include duplicates. Sort by last name then first name.
5. Create a report that returns the first and last names, emails, and phone numbers of all clients *who are also vendors*. Sort by last name then first name.
6. Create a report that returns the first and last names, emails, and phone numbers of all clients *except those that are also vendors*. Sort by last name then first name.
7. Create a report that returns the first and last names, emails, and phone numbers of all vendors *except those that are also clients*. Sort by last name then first name.

Solution: Set-Operators/Solutions/set-operators.sql

```
1.   -- Number of clients
2.   SELECT COUNT(*) FROM clients; -- 128
3.
4.   -- Number of vendors
5.   SELECT COUNT(*) FROM vendors; -- 90
6.
7.   -- All clients and vendors without duplicates
8.   SELECT first_name, last_name, email, phone_number
9.   FROM clients
10.  UNION
11.  SELECT first_name, last_name, email, phone_number
12.  FROM vendors
13.  ORDER BY last_name, first_name;
14.
15.  -- All clients and vendors with duplicates
16.  SELECT first_name, last_name, email, phone_number
17.  FROM clients
18.  UNION ALL
19.  SELECT first_name, last_name, email, phone_number
20.  FROM vendors
21.  ORDER BY last_name, first_name;
22.
23.  -- All clients who are also vendors
24.  SELECT first_name, last_name, email, phone_number
25.  FROM clients
26.  INTERSECT
27.  SELECT first_name, last_name, email, phone_number
28.  FROM vendors
29.  ORDER BY last_name, first_name;
30.
31.  -- All clients except those that are also vendors
32.  SELECT first_name, last_name, email, phone_number
33.  FROM clients
34.  MINUS
35.  SELECT first_name, last_name, email, phone_number
36.  FROM vendors
37.  ORDER BY last_name, first_name;
38.
39.  -- All vendors except those that are also clients
40.  SELECT first_name, last_name, email, phone_number
41.  FROM vendors
42.  MINUS
43.  SELECT first_name, last_name, email, phone_number
44.  FROM clients
```

```
45.  ORDER BY last_name, first_name;
```

8.6. Set Operators and Aliases

When using set operators with aliases, you only need to put the aliases on the first query segment. For example, the following will return the table name for several tables in the HR schema and the number of records in each table:

```
SELECT 'countries' AS table_name, COUNT(*) AS num_records
FROM countries
UNION
SELECT 'departments', COUNT(*)
FROM departments
UNION
SELECT 'employees', COUNT(*)
FROM employees
UNION
SELECT 'jobs', COUNT(*)
FROM jobs;
```


The output will look like this:

	TABLE_NAME	NUM_RECORDS
1	countries	25
2	departments	27
3	employees	108
4	jobs	19

Note that it is permissible to add the alias to each query segment as long as the aliases are the same, like this:

```
SELECT 'countries' AS table_name, COUNT(*) AS num_records
FROM countries
UNION
SELECT 'departments' AS table_name, COUNT(*) AS num_records
FROM departments
UNION
SELECT 'employees' AS table_name, COUNT(*) AS num_records
FROM employees
UNION
SELECT 'jobs' AS table_name, COUNT(*) AS num_records
FROM jobs;
```


In fact, because of a very strange bug in Oracle (bug details[3]), including aliases on each segment could save some headaches that occur when ordering by the alias.

Conclusion

In this lesson, you have learned to write queries using `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`.

3. https://stackoverflow.com/questions/25387951/curious-issue-with-oracle-union-and-order-by

LESSON 9
Conditional Processing with CASE

Topics Covered

☑ CASE.

Introduction

In this lesson, you will learn to use the CASE function to display different values depending on the values of a column or columns.

9.1. Using CASE

CASE statements are used to return different values based on the value of one or more expressions. They come in two types:

1. Simple or Selected Case
2. Searched Case

We will look at both types in this lesson.

9.2. Selected Case

A selected case statement evaluates a single expression and then compares it one-by-one to values in the WHEN clauses until it finds an exact match. When it does, it uses the return value in the WHEN's related THEN clause. If none of the values in the WHEN clauses is equal to the evaluated expression in the CASE clause, then the value in the ELSE clause is returned.

You must code END to terminate the scope of the case statement. Also, it's a good idea to assign an alias name to the case statement using AS.

The syntax for a selected case statement is shown below:

```
SELECT CASE expression
  WHEN value_1 THEN return_value_1
  WHEN value_2 THEN return_value_2
  WHEN value_3 THEN return_value_3
  WHEN value_4 THEN return_value_4
  ELSE return_value_default
END AS column_name
FROM table_name
```

Earlier, we used the following SQL statement to bucket sales representatives into teams:

Demo 9.1: Oracle-SQL-Functions/Solutions/mod.sql

```
1.   SELECT employee_id, first_name, last_name, hire_date,
2.     ROW_NUMBER() OVER (ORDER BY hire_date) AS row_num,
3.     MOD(ROW_NUMBER() OVER (ORDER BY hire_date), 3) AS team
4.   FROM employees
5.   WHERE job_id = 'SA_REP'
6.   ORDER BY hire_date;
```


Code Explanation

The output of this query isn't very satisfying:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	ROW_NUM	TEAM
1	156	Janette	King	30-JAN-04	1	1
2	157	Patrick	Sully	04-MAR-04	2	2
3	174	Ellen	Abel	11-MAY-04	3	0
4	158	Allan	McEwen	01-AUG-04	4	1
5	150	Peter	Tucker	30-JAN-05	5	2
6	159	Lindsey	Smith	10-MAR-05	6	0
7	168	Lisa	Ozer	11-MAR-05	7	1
8	175	Alyssa	Hutton	19-MAR-05	8	2
9	151	David	Bernstein	24-MAR-05	9	0
10	152	Peter	Hall	20-AUG-05	10	1

Rather than team values of 0, 1, and 2, it would be nice to give the teams fun names. We can use a selected case to do that:

Demo 9.2: Case/Demos/selected-case.sql

```
1.   SELECT employee_id, first_name, last_name, hire_date,
2.     ROW_NUMBER() OVER (ORDER BY hire_date) AS row_num,
3.     CASE MOD(ROW_NUMBER() OVER (ORDER BY hire_date), 3)
4.       WHEN 1 THEN 'Tigers'
5.       WHEN 2 THEN 'Eagles'
6.       ELSE 'Dolphins'
7.     END AS team
8.   FROM employees
9.   WHERE job_id = 'SA_REP'
10.  ORDER BY hire_date;
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	ROW_NUM	TEAM
1	156	Janette	King	30-JAN-04	1	Tigers
2	157	Patrick	Sully	04-MAR-04	2	Eagles
3	174	Ellen	Abel	11-MAY-04	3	Dolphins
4	158	Allan	McEwen	01-AUG-04	4	Tigers
5	150	Peter	Tucker	30-JAN-05	5	Eagles
6	159	Lindsey	Smith	10-MAR-05	6	Dolphins
7	168	Lisa	Ozer	11-MAR-05	7	Tigers
8	175	Alyssa	Hutton	19-MAR-05	8	Eagles
9	151	David	Bernstein	24-MAR-05	9	Dolphins
10	152	Peter	Hall	20-AUG-05	10	Tigers

9.3. Searched Case

A searched case statement evaluates expressions in the `WHEN` clauses one-by-one until it finds a true expression. When it does, it uses the return value in the `WHEN`'s related `THEN` clause. If none of the expressions in the `WHEN` clauses evaluate to true, then the value in the `ELSE` clause is returned.

The syntax for a searched case statement is shown below:

Searched Case Syntax

```
SELECT CASE
  WHEN expression_1 THEN return_value_1
  WHEN expression_2 THEN return_value_2
  WHEN expression_3 THEN return_value_3
  WHEN expression_4 THEN return_value_4
  ELSE return_value_default
END AS column_name
FROM table
```

We can calculate the midpoint between the `min_salary` and the `max_salary` in the `jobs` table using the formula below:

```
(max_salary + min_salary) / 2
```

By comparing employees' salaries to this midpoint, we can see who earns more or less than the midpoint. We can use a searched case statement with this formula to output 'Above Midpoint', 'Below Midpoint', or 'Midpoint' for each employee.

Demo 9.3: Case/Demos/searched-case.sql

```
1.   SELECT e.first_name, e.last_name, e.salary, j.job_title,
2.     CASE
3.       WHEN e.salary > (j.max_salary + j.min_salary) / 2
4.         THEN 'Above Midpoint'
5.       WHEN e.salary < (j.max_salary + j.min_salary) / 2
6.         THEN 'Below Midpoint'
7.       ELSE 'Midpoint'
8.     END AS salary_status
9.   FROM employees e
10.    JOIN jobs j ON j.job_id = e.job_id
11.  ORDER BY e.last_name, e.first_name;
```

Code Explanation

The above `SELECT` statement will return the following results (first 10 rows shown):

	FIRST_NAME	LAST_NAME	SALARY	JOB_TITLE	SALARY_STATUS
1	Ellen	Abel	11000	Sales Representative	Above Midpoint
2	Sundar	Ande	6400	Sales Representative	Below Midpoint
3	Mozhe	Atkinson	2800	Stock Clerk	Below Midpoint
4	David	Austin	4800	Programmer	Below Midpoint
5	Hermann	Baer	10000	Public Relations Representative	Above Midpoint
6	Shelli	Baida	2900	Purchasing Clerk	Below Midpoint
7	Amit	Banda	6200	Sales Representative	Below Midpoint
8	Elizabeth	Bates	7300	Sales Representative	Below Midpoint
9	Sarah	Bell	4000	Shipping Clerk	Midpoint
10	David	Bernstein	9500	Sales Representative	Above Midpoint

Exercise 25: Working with CASE

15 to 25 minutes

In this exercise you will practice using `CASE`.

You can use the following query to get the number of employees by department:

```
SELECT d.department_name, COUNT(e.employee_id) AS num_employees
FROM employees e
  RIGHT JOIN departments d ON d.department_id = e.department_id
GROUP BY d.department_name
ORDER BY num_employees DESC;
```

1. Open a new SQL Worksheet in SQL Developer and save it as `case.sql` in `Case/Exercises`.
2. Starting with the query above, add a column with the alias `department_size` that outputs:
   - 'Large' if the department has at least 10 employees.
   - 'Medium' if the department has at least 5 but fewer than 10 employees.
   - 'Small' if the department has at least 1 but fewer than 5 employees.
   - 'Empty' if the department has no employees.

The first 12 rows of the results from the solution are shown below:

	DEPARTMENT_NAME	NUM_EMPLOYEES	DEPARTMENT_SIZE
1	Shipping	45	Large
2	Sales	34	Large
3	Purchasing	6	Medium
4	Finance	6	Medium
5	IT	5	Medium
6	Executive	3	Small
7	Accounting	2	Small
8	Marketing	2	Small
9	Public Relations	1	Small
10	Human Resources	1	Small
11	Administration	1	Small
12	Payroll	0	Empty

Solution: Case/Solutions/case.sql

```
1.   SELECT d.department_name, COUNT(e.employee_id) AS num_employees,
2.     CASE
3.       WHEN COUNT(e.employee_id) >= 10 THEN 'Large'
4.       WHEN COUNT(e.employee_id) >= 5 THEN 'Medium'
5.       WHEN COUNT(e.employee_id) >= 1 THEN 'Small'
6.       ELSE 'Empty'
7.     END AS department_size
8.   FROM employees e
9.     RIGHT JOIN departments d ON d.department_id = e.department_id
10.  GROUP BY d.department_name
11.  ORDER BY num_employees DESC;
```


Conclusion

In this lesson, you learned about using `CASE` to output different values in reports based on data contained in table fields.

LESSON 10
Data Manipulation Language

Topics Covered

☑ `INSERT`.

☑ `UPDATE`.

☑ `DELETE`.

Introduction

The Data Manipulation Language (DML) is used to add, modify, and remove data in existing tables within a schema. In this lesson, we will cover `INSERT`, `UPDATE`, and `DELETE` statements. Inserting new records into a table is not difficult. Dangerously, it is even easier to update and delete records. Note that the database administrator can block developers from being able to use these statements.

Resetting the HR Schema

If at any point, you want to reset the **HR** schema back to its original structure and content, you can do so by opening and executing the `setup/reset-hr-schema.sql` file in your class files.

10.1. Transactions and Sessions

When you open SQL Developer and make a connection, a session starts, and along with it, a new transaction. By default, any inserts, updates, or deletes that you do during the session can be rolled back until you either run a Database Definition Language (DDL) statement (e.g., `CREATE TABLE`) or explicitly commit them by running this simple command:

```
COMMIT;
```

This will end the current transaction and start a new one.

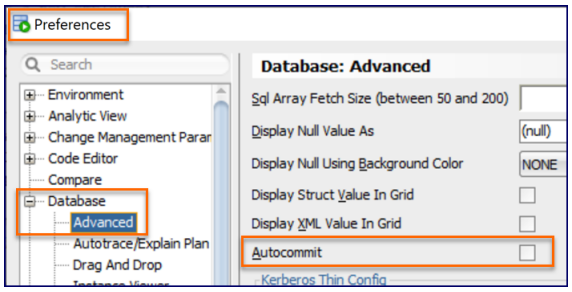
Until your changes are committed, they will not be visible to others accessing the database. This can cause issues as your pending changes might prevent another developer from accessing or modifying a table. As a rule, you should commit your changes as soon as you are sure of them.

At any point you can run the following command to undo any changes you have made during the current transaction (i.e., since the last commit):

```
ROLLBACK;
```

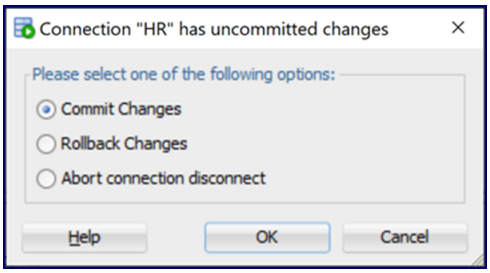
In SQL Developer, you can set your preference for autocommitting by:

1. Selecting **Preferences** from the **Tools** menu.
2. Selecting **Database > Advanced**.
3. Checking or unchecking **Autocommit**.



We recommend that you **DO NOT** check **Autocommit**.

If you do not have **Autocommit** turned on, when you close SQL Developer, if you have any uncommitted changes you will be prompted to commit them or roll them back:



Resetting the HR Schema

If at any point, you want to reset the **HR** schema back to its original structure and content, you can do so by opening and executing the `setup/reset-hr-schema.sql` file in your class files.

10.2. INSERT

To insert a record into a table, you must specify values for all fields that do not have default values and cannot be `NULL`.

```
INSERT INTO table_name
(column_names)
VALUES (values);
```

The second line of the above statement can be excluded if all required columns are inserted and the values are listed in the same order as the columns in the table. We recommend you include the second line all the time though as the code will be easier to read and update and less likely to break as the database is modified.

Demo 10.1: Data-Manipulation-Language/Demos/insert.sql

```
1.   INSERT INTO clients
2.   (first_name, last_name, email, phone_number)
3.   VALUES ('Serena', 'Williams', 'SWILLIAMS', '555.843.2378');
4.
5.   INSERT INTO vendors
6.   (first_name, last_name, email, phone_number)
7.   VALUES ('Serena', 'Williams', 'SWILLIAMS', '555.843.2378');
```

Code Explanation

If the `INSERT` is successful, the output will read something to this effect:

```
1 row inserted.
```

You can then query the table to see the new row:

```
SELECT *
FROM clients
ORDER BY client_id DESC
FETCH FIRST 1 ROW ONLY;
```

Auto-incrementing Primary Keys

The clients and vendors tables were created using auto-incrementing primary keys:

```
CREATE TABLE clients
(
  client_id NUMBER GENERATED AS IDENTITY,...
```

As such, attempting to insert an explicit value for the primary key column will result in an error.

Exercise 26: Inserting Records

5 to 10 minutes

Open a new SQL Worksheet in SQL Developer and save it as `insert.sql` in `Data-Manipulation-Language/Exercises`.

Insert yourself into the `clients` and `vendors` tables.

Solution: Data-Manipulation-Language/Solutions/insert.sql

```
1.   INSERT INTO clients
2.   (first_name, last_name, email, phone_number)
3.   VALUES ('Serena', 'Williams', 'SWILLIAMS', '555.843.2378');
4.
5.   INSERT INTO vendors
6.   (first_name, last_name, email, phone_number)
7.   VALUES ('Serena', 'Williams', 'SWILLIAMS', '555.843.2378');
```

Code Explanation

This solution assumes that your name is Serena Williams.

10.3. UPDATE

The `UPDATE` statement allows you to update one or more fields for any number of records in a table. *You must be very careful not to update more records than you intend to!*

```
UPDATE table_name
SET column_name = value,
  column_name = value,
  column_name = value
WHERE conditions;
```

Demo 10.2: Data-Manipulation-Language/Demos/update.sql

```
1.   -- First, find my client_id
2.   SELECT *
3.   FROM clients
4.   WHERE first_name = 'Nat' AND last_name = 'Dunn';
5.
6.   -- Now update using that id
7.   UPDATE clients
8.   SET first_name = 'Nathaniel',
9.     email = 'NATDUNN'
10.  WHERE client_id = 129;
```

Code Explanation

While it is possible to use other conditions in the `WHERE` clause to find the records to update, using the primary key provides a safe way to ensure that you are only updating one record.

If the `UPDATE` is successful, the output will read something to this effect:

```
1 row updated.
```

Don't Forget the WHERE Clause!

If you leave off the `WHERE` clause in an update statement, you will update every record in the table. For example, the following statement would rename all clients 'Nathaniel':

```
UPDATE clients
SET first_name = 'Nathaniel';
```


Be careful! As Spiderman's Uncle Ben says, *"With great power comes great responsibility."*

10.4. DELETE

The `DELETE` statement allows you to delete one or more records in a table. *Like with `UPDATE`, you must be very careful not to delete more records than you intend to!*

```
DELETE FROM table_name
      WHERE conditions;
```

Demo 10.3: Data-Manipulation-Language/Demos/delete.sql

```
1.    -- First, find my client_id
2.    SELECT *
3.    FROM clients
4.    WHERE first_name = 'Nathaniel' AND last_name = 'Dunn';
5.
6.    -- Now delete using that id
7.    DELETE FROM clients
8.    WHERE client_id = 129;
```

Code Explanation

If the `DELETE` is successful, the output will read something to this effect:

```
1 row deleted.
```


Don't Forget the WHERE Clause!

If you leave off the `WHERE` clause in an delete statement, you will delete every record in the table. Don't do this:

```
DELETE FROM clients;
```

...unless, of course, your intention is to empty the `clients` table.

Exercise 27: Updating and Deleting Records

5 to 15 minutes

In this exercise, you will practice updating and deleting records.

1. Open a new SQL Worksheet in SQL Developer and save it as `update-delete.sql` in `Data-Manipulation-Language/Exercises`.
2. Update your record in the `clients` and/or `vendors` in any way you like.
3. Delete your record from the `clients` and/or `vendors`.

Solution: Data-Manipulation-Language/Solutions/update-delete.sql

```
1.   -- CLIENTS TABLE
2.   -- First, find your client id
3.   SELECT *
4.   FROM clients
5.   WHERE first_name = 'Serena' AND last_name = 'Williams';
6.
7.   -- Update your record
8.   UPDATE clients
9.   SET phone_number = '555.293.7663',
10.    email = 'SERENA'
11.  WHERE client_id = 130;
12.
13.  -- See your update
14.  SELECT *
15.  FROM clients
16.  WHERE client_id = 130;
17.
18.  -- VENDORS TABLE
19.  -- First, find your vendor id
20.  SELECT *
21.  FROM vendors
22.  WHERE first_name = 'Serena' AND last_name = 'Williams';
23.
24.  -- Update your record
25.  UPDATE vendors
26.  SET phone_number = '555.293.7663',
27.    email = 'SERENA'
28.  WHERE vendor_id = 92;
29.
30.  -- See your update
31.  SELECT *
32.  FROM vendors
33.  WHERE vendor_id = 92;
```


10.5. Updating and Deleting Multiple Records

There are times when you want to update or delete multiple records at one fell swoop. That's not hard to do. For example, the following query would prepend "1." to all phone numbers in the `clients` table:

```
UPDATE clients
SET phone_number = '1.' || phone_number;
```

If successful, the output will read something to this effect:

```
129 rows updated.
```

And the following query would delete all clients for whom we have no contact information:


```
DELETE FROM clients
WHERE phone_number IS NULL AND email IS NULL;
```

Again, **be careful with these types of updates and deletes**. It's mighty easy to make big mistakes.

Conclusion

In this lesson, you have learned how to insert, update, and delete records. Remember to be careful with updates and deletes. If you forget or mistype the `WHERE` clause, you could lose a lot of data.

LESSON 11
Creating Views

Topics Covered

☑ What are views?

☑ `CREATE VIEW`.

☑ `DROP VIEW`.

Introduction

Views are essentially saved `SELECT` queries that can themselves be queried. They are used to provide easier access to data.

11.1. Creating Views

To create views, simply write your `SELECT` query and wrap it in a `CREATE VIEW` statement as shown below:

```
CREATE VIEW view_name AS
SELECT statement goes here...
```

For example, to show a report that provides employee names along with their managers' names, you must do a self join like this:

```
CREATE VIEW managers_view AS
SELECT e.first_name || ' ' || e.last_name AS employee,
  m.first_name || ' ' || m.last_name AS manager
FROM employees e
  LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

The following example shows how to turn this into a view:

Demo 11.1: Creating-Views/Demos/create-view.sql

```
1.   CREATE VIEW managers_view AS
2.   SELECT e.first_name || ' ' || e.last_name AS employee,
3.          m.first_name || ' ' || m.last_name AS manager
4.   FROM employees e
5.       LEFT JOIN employees m ON e.manager_id = m.employee_id;
6.
7.   -- After creating the view, you can query it as if it were a table
8.   SELECT *
9.   FROM managers_view;
```

Code Explanation

You can see that the process of creating the view is simple. Also notice how you can query the view as if it were a table.

❖ 11.1.1. Dropping Views

Don't like your view? Dropping it is easy enough:

```
DROP VIEW view_name
```


11.2. Benefits of Views

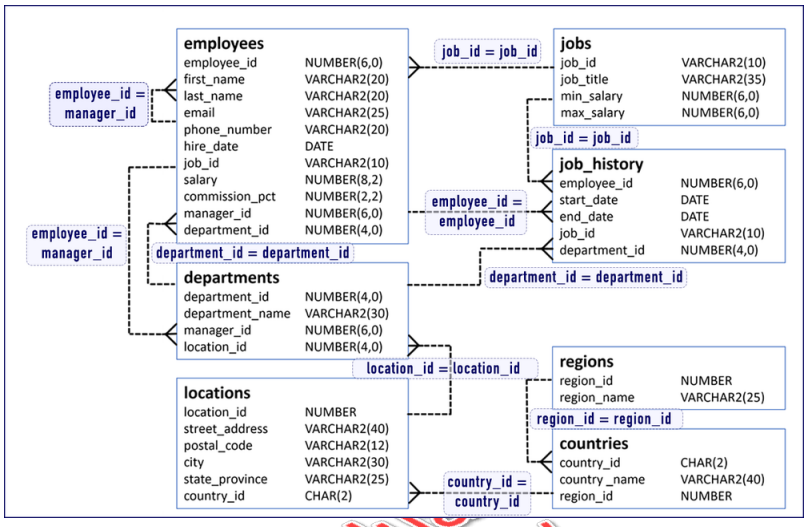
Views have the following benefits:

- *Security* - Views can be made accessible to users while the underlying tables are not directly accessible. This allows the DBA to give users only the data they need, while protecting other data in the same table.
- *Simplicity* - Views can be used to hide and reuse complex queries.
- *Column Name Simplication or Clarification* - Views can be used to provide aliases on column names to make them more memorable and/or meaningful.
- *Stepping Stone* - Views can provide a stepping stone in a "multi-level" query. For example, you could create a view of a query that counted the number of sales each salesperson had made. You could then query that view to group the salespeople by the number of sales they had made.

Exercise 28: Creating a View

25 to 40 minutes

In this exercise you will create a view that shows extensive details about employees. You may find it useful to refer to the entity diagram below:



1. Open a new query window and create a view called `employee_info_view` that has the following columns:

   - `employee_id` - The employee id of the employee.
   - `first_name` - The first name of the employee.
   - `last_name` - The last name of the employee.
   - `salary` - The salary of the employee.
   - `commission_pct` - The commission percentage (as a decimal) of the employee.
   - `manager_id` - The employee id of the employee's manager.
   - `mgr_first_name` - The first name of the employee's manager.
   - `mgr_last_name` - The last name of the employee's manager.
   - `job_id` - The job id of the employee.
   - `job_title` - The job id of the employee.
   - `department_id` - The department id of the employee's department.
   - `department_name` - The department name of the employee's department.
   - `location_id` - The location id of the employee's department.

- `city` - The city in which the employee's department is located.
- `state_province` - The state in which the employee's department is located.
- `country_id` - The id of the country in which the employee's department is located.
- `country_name` - The name of the country in which the employee's department is located.
- `region_name` - The name of the region in which the employee's department is located.

2. Write a query using this view to get `first_name`, `last_name`, `salary`, `commission_pct`, `mgr_first_name`, `mgr_last_name`, `city`, `state_province`, and `country_name` for all employees who earn a commission sorted by salary from highest to lowest. The first 10 results are shown below:

	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT	MGR_FIRST_NAME	MGR_LAST_NAME	CITY	STATE_PROVINCE	COUNTRY_NAME
1	John	Russell	14000	0.4	Steven	King	Oxford	Oxford	United Kingdom
2	Karen	Partners	13500	0.3	Steven	King	Oxford	Oxford	United Kingdom
3	Alberto	Errazuriz	12000	0.3	Steven	King	Oxford	Oxford	United Kingdom
4	Lisa	Ozer	11500	0.25	Gerald	Cambrault	Oxford	Oxford	United Kingdom
5	Ellen	Abel	11000	0.3	Eleni	Zlotkey	Oxford	Oxford	United Kingdom
6	Gerald	Cambrault	11000	0.3	Steven	King	Oxford	Oxford	United Kingdom
7	Eleni	Zlotkey	10500	0.2	Steven	King	Oxford	Oxford	United Kingdom
8	Clara	Vishney	10500	0.25	Alberto	Errazuriz	Oxford	Oxford	United Kingdom
9	Harrison	Bloom	10000	0.2	Gerald	Cambrault	Oxford	Oxford	United Kingdom
10	Peter	Tucker	10000	0.3	John	Russell	Oxford	Oxford	United Kingdom

3. Save your solution as `create-view.sql` in `Creating-Views/Exercises`.

Solution: Creating-Views/Solutions/create-view.sql

```
1.   CREATE VIEW employee_info_view AS
2.   SELECT e.employee_id, e.first_name, e.last_name,
3.    e.salary, e.commission_pct, e.manager_id,
4.     m.first_name AS mgr_first_name, m.last_name AS mgr_last_name,
5.    j.job_id, j.job_title,
6.    d.department_id, d.department_name,
7.    l.location_id, l.city, l.state_province,
8.    c.country_id, c.country_name,
9.    r.region_name
10.  FROM employees e
11.    JOIN employees m ON m.employee_id = e.manager_id
12.   JOIN departments d ON e.department_id = d.department_id
13.   JOIN jobs j ON j.job_id = e.job_id
14.   JOIN locations l ON d.location_id = l.location_id
15.   JOIN countries c ON l.country_id = c.country_id
16.   JOIN regions r ON c.region_id = r.region_id;
17.
18.  SELECT first_name, last_name, salary, commission_pct,
19.    mgr_first_name, mgr_last_name,
20.   city, state_province, country_name
21.  FROM employee_info_view
22.  WHERE commission_pct IS NOT NULL
23.  ORDER BY salary DESC;
```


11.3. Inline Views

All of the demo queries in this section are in the `Creating-Views/Demos/inline-view.sql` file.

Please open that file in SQL Developer to follow along with the demos.

An inline view is a subquery that is used as a table. It is like a regular view except that it is not stored in the schema; it only used for the query that it is within.

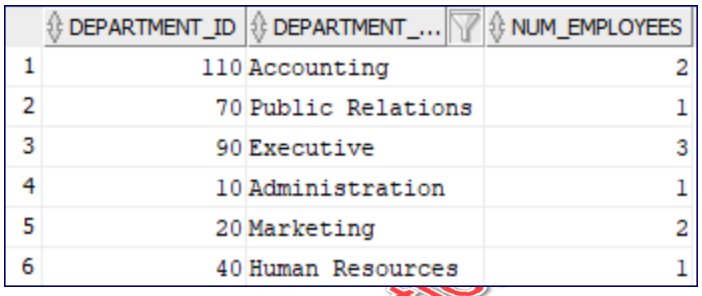
Consider the following challenge: Your manager has asked you how many employees work in departments with three or fewer employees in them.

You can do this in two steps:

1. Find all the department ids for departments that have three or fewer employees:

```
SELECT d.department_id, d.department_name,
  (e.employee_id) AS num_employees
FROM departments d
  JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_id, d.department_name
HAVING COUNT(e.employee_id) <= 3;
```

This will return:



	DEPARTMENT_ID	DEPARTMENT_...	NUM_EMPLOYEES
1	110	Accounting	2
2	70	Public Relations	1
3	90	Executive	3
4	10	Administration	1
5	20	Marketing	2
6	40	Human Resources	1

2. Count them up:

```
SELECT COUNT(employee_id) AS num_employees
FROM employees
WHERE department_id IN (110, 70, 90, 10, 20, 40);
```

But that's a lot of work and not easy to modify, if say, your manager then wants to know how many employees are in departments with *four or fewer employees*.

Instead, you can create a query like the one below to be used as an inline view:

```
SELECT d.department_name,
  COUNT(e.employee_id) AS num_employees
FROM departments d
  JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_name;
```

Here's the result set:

	DEPARTMENT_NAME	NUM_EMPLOYEES
1	Sales	34
2	Marketing	2
3	Administration	1
4	Purchasing	6
5	Shipping	45
6	IT	5
7	Executive	3
8	Finance	6
9	Public Relations	1
10	Human Resources	1
11	Accounting	2

You can treat this result set like a table by nesting the query in another query:

```
SELECT SUM(num_employees) AS num_employees
FROM
(
  SELECT d.department_name,
    COUNT(e.employee_id) AS num_employees
  FROM departments d
    JOIN employees e ON d.department_id = e.department_id
  GROUP BY d.department_name
)
WHERE num_employees <= 3;
```


And there you have an inline view. Now it's easy to get the number of employees in different sized departments by simply changing the number in the `WHERE` clause.

A really useful inline view might be a good candidate for a permanent view.

Conclusion

In this lesson, you have learned the purpose and benefits of views and how to create and drop them.