Introduction to SQL

7400 E. Orchard Road, Suite 1450 N, Greenwood Village, CO 80111
303-302-5234 | 800-292-3716
SkillDistillery.com

Introduction to SQL

Student Workbook

itcourseware
Training Materials For IT Professionals

INTRODUCTION SQL

Author: Rob Roselius

Published by ITCourseware, LLC, 7400 E. Orchard Road, Suite 1450, Greenwood Village, CO 80111

Editor: Jan Waleri

Editorial Assistant: Ginny Jaranowski

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Elizabeth Boss, Denise Geller, Jennifer James, Julie Johnson, Roger Jones, John McCallister, Joe McGlynn, Jim McNally, Kevin Smith, Danielle Waleri and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

SQL Queries – The SELECT Statement

Objectives

- Retrieve data from a table using the SQL **SELECT** statement.
- Filter the returned rows with a **WHERE** clause.
- Locate fields which have not yet been populated.
- Combine multiple search criteria in a **WHERE** clause using logical operators.
- Use the **IN** and **BETWEEN** operators to match a column against multiple values.
- Use wildcards to search for a pattern in character data.
- Sort the output of a **SELECT** statement.

THE SELECT STATEMENT

- SQL data retrievals are done using the **SELECT** statement.
  - Data retrievals are also called *queries*.
- A **SELECT** statement may make use of several *clauses*, but minimally must include the following two:

```
SELECT [DISTINCT] {item_list | *}
  FROM table;
```

- The **SELECT** clause specifies which data values will be retrieved, and will be followed by either an ***item_list*** or a *****.
  - ***** represents all the columns in ***table***.
  - ***item_list*** is a column, an expression, or a comma-separated list of any combination of these.
  - The query will return each item in ***item_list*** under a column heading of the same name, unless an alias is specified.
  - Use **DISTINCT** to eliminate duplicate rows from the displayed results.
- The **FROM** clause specifies one or more sources, typically tables, from which to retrieve data.
- A heading may be overridden with a *column alias,* a word used to replace the default heading.

```
SELECT name, minimum_salary,
       minimum_salary / maximum_salary AS pay_ratio
  FROM jobs;
```


The **SELECT** statement may query one or many tables. The data that is retrieved may be thought of as a *virtual table* (i.e., one that exists in memory only until the query completes). This virtual table is also referred to as the query's *result set*.

The **SELECT** statement can be used to retrieve data from any or all columns from a table. For example, the following **SELECT** will display a list of everything about every job in the database:

```
SELECT *
  FROM jobs;
```

The query below will list the minimum salary for each job:

```
SELECT minimum_salary
  FROM jobs;
```

The query below will list each minimum salary once:

```
SELECT DISTINCT minimum_salary
  FROM jobs;
```

This query will retrieve each job's name and pay limits.

```
SELECT name, minimum_salary, maximum_salary
  FROM jobs;
```

Note:

Table, column, or alias names containing spaces must be enclosed in quotes.

```
SELECT name, minimum_salary,
       minimum_salary / maximum_salary AS "Pay Ratio"
  FROM jobs;
```

Choosing Rows with the WHERE Clause

- Use the **WHERE** clause when only a subset of all the rows in a table is required.

```
SELECT {item_list | *}
  FROM table
[WHERE conditions];
```

- The **WHERE** clause is evaluated once for each row in ***table*.**

- ***conditions*** will be one or more expressions that will evaluate to either true, false, or neither (null).

  - If the **WHERE** clause evaluates to true, then the current row is returned in the result set.

- Operators that may be used in the **WHERE** clause include:

=	equal to
!=, ^=, <>	not equal to
>, <	greater than, less than
>=, <=	greater than or equal to, less than or equal to

- Any character string values in conditions must be enclosed within single quotes.

  - Database data inside single quotes is case sensitive.

Retrieve all information for each department at location 3:

```
SELECT *
  FROM departments
 WHERE location_id = 3;
```

Retrieve first and last name, salary, and hire date for employees in Massachusetts:

```
SELECT firstname, lastname, salary, hiredate
  FROM employees
 WHERE state = 'MA';
```

Despite well-established standards, database products vary in certain low-level details of syntax. In ANSI standard SQL, doublequotes enclose identifiers - table, column, or alias names - *only*, and string literals must be enclosed in single-quotes. Oracle Database enforces this, but MySQL treats single, double, and back quotes interchangably (unless a standards-compliance mode is enabled.)

Simiilarly, standard SQL comments have two forms:

* C-style block comments:

```
SELECT id
       , firstname
       , lastname
    /* , commission_pct
       , salary          */
  FROM employees;
```

* To-end-of-line comments:

```
SELECT id, commission_pct, -- commision_pct is an integer value
       salary
  FROM employees;
```

However, MySQL requires at least one space to follow the --, and also supports # as a to-end-of-line comment starter.

Best practice is always to code to the established standards.

NULL Values

- A *null* is the absence of a value.
  - Null values are not equal to **0**, blanks, or empty strings.
- Compare null values with the **IS** operator.

```
SELECT *
  FROM projects
 WHERE end_date IS NULL;
```

  - The **IS** operator will result in either true or false.
- To search for any non-null values, use **IS NOT NULL**.

```
SELECT *
  FROM projects
 WHERE end_date IS NOT NULL;
```

- **NULL** is not the same as a "false" value, but it isn't a "true" value, either.
  - **NULL** means "unknown;" any operation with a **NULL** (other than **IS** or **IS NOT**) yields a **NULL**.

```
SELECT *
  FROM projects
 WHERE end_date = NULL;
```

  - This will never be true, even for rows with null values in the **end_date** column!

Retrieve all employees with a known commission percent:

```
SELECT lastname, firstname, salary, commission_pct
  FROM employees
 WHERE commission_pct IS NOT NULL;
```

Retrieve all employees with a commission percent that is either unknown, or 0:

```
SELECT lastname, firstname, salary, commission_pct
  FROM employees
 WHERE commission_pct IS NULL
       OR commission_pct = 0;
```

VARCHAR, VARCHAR2, and NULL

The SQL standards specify that for character-varying data an empty string, `''`, is a distinct, non-null value and should not be treated as a NULL. For example, if we know a person does not have a middle name, we would store empty string (`''`) in their middle name column; but if we simply don't yet know what their middle name is, or whether they have one at all, we would store `NULL`. We should then get different results for the following two queries:

```
SELECT * FROM employees WHERE middlename = '';
SELECT * FROM employees WHERE middlename IS NULL;
```

Oracle's character-varying datatype does not comply with this, so Oracle named it VARCHAR2 instead of VARCHAR. Oracle VARCHAR2 treats empty string and NULL identically. Oracle reserves the word VARCHAR for a future implementation that complies with the SQL standards. However, it has never implemented this, so you use only VARCHAR2 with Oracle.

Compound Expressions

- Use logical operators to group conditions in a **WHERE** clause.
  - **NOT** will negate the condition following it.
  - **AND** will result in true if both conditions are true for a row.
  - **OR** will result in true if either condition is true for a row.
- The logical operators are evaluated based on precedence: **NOT** has the highest precedence, then **AND**, and then **OR**.
  - If a **WHERE** clause contains both an **AND** and an **OR**, the **AND** will always be evaluated first.

```
SELECT id, firstname, lastname
  FROM employees
 WHERE firstname = 'John'
    OR firstname = 'Katelyn'
   AND lastname = 'Smith';
```

    - This query will find **Katelyn Smith,** as well as anyone with the first name of **John**.

- Use parentheses to override the precedence of these operators.

```
SELECT id, firstname, lastname
       FROM employees
      WHERE ( firstname = 'John'
         OR firstname = 'Katelyn' )
        AND lastname = 'Smith';
```

  - This query will find all people with a **lastname** of **Smith** and a **firstname** of either **John** or **Katelyn**.

When an employee enters data into the system, a field for which there is no data may be skipped or have a **0** entered.

```
SELECT lastname, firstname, salary, commission_pct
  FROM employees
 WHERE state = 'MA'
       AND ( commission_pct IS NULL
             OR commission_pct = 0 );
```

IN AND BETWEEN

- Use the **IN** (or **NOT IN**) operator to compare a column against several possible values.

```
SELECT lastname, firstname, state
  FROM employees
 WHERE state IN ('GA', 'NC');
```

  - **IN** is equivalent to **OR**ing several conditions together.

```
SELECT lastname, firstname, state
  FROM employees
 WHERE state = 'GA'
    OR state = 'NC';
```

- Use the **BETWEEN** operator to compare a column against a range of inclusive values.

```
SELECT id, lastname, lastname
  FROM employees
 WHERE id BETWEEN 50 AND 70;
```

  - The **AND** is part of the **BETWEEN** operator.

  - Make sure that lower value appears before the higher one:

```
SELECT id, lastname, hiredate
  FROM employees
 WHERE hiredate BETWEEN '1990-01-01' AND '2000-01-01';
```


Retrieve a list of New Jersey employees who don't live in the 08538 or 08561 zip codes:

```
SELECT lastname, city, state, zipcode
  FROM employees
 WHERE zipcode NOT IN ('08538', '08561')
   AND state = 'NJ';
```

List employees of department 6 who make between $30,000 and $40,000 inclusive:

```
SELECT id, lastname, salary
  FROM employees
 WHERE department_id = 6 AND salary BETWEEN 30000 AND 40000;
```

PATTERN MATCHING: LIKE AND REGEXP

- The **LIKE** operator provides pattern matching for character data with two simple *wildcards*:
  - **%** Matches zero or more characters.
  - _ Matches exactly one character and is position-dependent.

```
SELECT * FROM jobs
 WHERE name LIKE 'Senior%';

SELECT * FROM jobs
 WHERE name LIKE '%Administrator';

SELECT * FROM jobs
 WHERE name LIKE '%Admin%';

SELECT firstname, lastname FROM employees
 WHERE lastname LIKE 'P____';

SELECT lastname FROM employees
 WHERE lastname LIKE '%son';
```

- Many database vendors support Regular Expression pattern matching, though the syntax varies.
  - MySQL:

```
SELECT lastname FROM employees
 WHERE lastname REGEXP '[WPR].*ert?son';
```

  - Oracle Database:

```
SELECT lastname FROM employees
 WHERE REGEXP_LIKE(lastname, '[WPR].*ert?son');
```


SQL Functions

Every database product supports its own list of *scalar* or *single-row functions* - functions that can be used in SQL statements and which are evaluated for every row.

String Functions:

```
SELECT CONCAT(lastname, ', ', firstname) FROM employees;
SELECT lastname, LENGTH(lastname) FROM employees;
SELECT name, LOWER(name) FROM jobs;
SELECT SUBSTR(lastname, 1, 3) FROM employees;
SELECT name, INSTR(name, ' ') FROM jobs;
SELECT name, SUBSTR(name, 1, INSTR(name, ' ')) FROM jobs;
```

Numeric Functions:

```
SELECT COS(1);
SELECT 4 * ATAN(1);
SELECT SQRT(2);
SELECT FLOOR(RAND() * 52);
SELECT  minimum_salary/maximum_salary*100,
        ROUND(minimum_salary/maximum_salary*100,2) FROM jobs;
```

Date and Time Functions:

```
SELECT CURDATE(), CURTIME(), NOW();
SELECT start_date, end_date, DATEDIFF(end_date, start_date)
  FROM assignments;
SELECT lastname, MONTH(hiredate) FROM employees;
SELECT DATE_ADD(CURDATE(), INTERVAL 30 DAY);
SELECT DATE_ADD(CURDATE(), INTERVAL 6 MONTH);
SELECT WEEKDAY(DATE_ADD(CURDATE(), INTERVAL 6 MONTH));
```

System and Information Functions:

```
SELECT USER();
SELECT DATABASE();
SELECT VERSION();
```

These functions are not standardized, though most vendors support a rich set of functions in similar categories. You must refer to your particular database product's documentation for function names and usage.

CREATING SOME ORDER

- Data is not physically stored in a table in any specified order.
  - Unless the table is an Index-Organized Table (IOT), records are usually appended to the end of the table.
- The **SELECT** statement's **ORDER BY** clause is the only way to enforce sequencing on the result set.
- The **ORDER BY** clause may be included in the **SELECT** statement:

```
   SELECT {item_list | *}
     FROM table
[ORDER BY column [ASC | DESC],...];
```

  - ***column*** can either be the name or the numeric position of the column in the ***item_list***.
- The result set is sorted by the first column name in the **ORDER BY** clause.
  - If there are duplicate values in the first sort column, the sort is repeated on the second column in the list, and so on.
  - The rows may be sorted in ascending (the default) or descending order.
  - **ASC** and **DESC** may be applied independently to each column.
  - **NULL** values will be sorted after all non-**NULL** values in **ASC** order (and before all non-**NULL**s when you use **DESC**).

Retreive a list of employees sorted by state from lowest to highest, and by salary within each state from highest to lowest:

```
SELECT id, firstname, lastname, salary, state
  FROM employees
 ORDER BY state ASC, salary DESC;
```

If all employees were to receive an 8% raise, show what the amount would be:

```
SELECT id, lastname, salary, salary * .08
  FROM employees
 ORDER BY 4 DESC;
```

An equivalent, more maintainable, and usually more readable, form of the above makes use of column aliases:

```
SELECT id, lastname, salary, salary * .08 AS proposed_raise
  FROM employees
 ORDER BY proposed_raise DESC;
```

LABS

Write queries to:

❶ Retrieve a list of all departments and their location ids.

❷ List the states in which there are employees, listing each state only once.

❸ List the states in which there are people, listing each state only once.

❹ List the employees of department 4 who live in North Carolina.

❺ List the jobs whose names end in "Engineer" or "Developer" or "Administrator".

❻ Show which projects, if any, have no end dates.

SQL Queries – Joins

Objectives

- Collect data from multiple tables with a single query.
- Use the relational aspects of your database in queries.
- Relate records within the same table.
- Use **LEFT**, **RIGHT**, and **FULL** outer joins.

SELECTING FROM MULTIPLE TABLES

- To retrieve data from more than one table in the same **SELECT** statement, the tables are *joined* together.

- Specify the tables in the **FROM** clause using the **INNER JOIN** keywords.

  - Use the **ON** clause with a join condition to specify how the rows in the second table should match the original rows.

```
SELECT lastname, name
  FROM employees INNER JOIN departments
          ON employees.department_id = departments.id;
```

  - The word **INNER** is optional.

- If you reference a column whose name appears in more than one table, you must precede the column name with the table name, called a *qualified reference*.

```
SELECT employees.id, lastname, name ...
```

  - Using a *table alias* can save you some typing.

```
SELECT e.id, e.lastname, d.name
  FROM employees e JOIN departments d
                       ON e.department_id = d.id;
```

- If the join clause involves only columns with the same name, you may use the **USING** clause, omitting the explicit join condition:

```
SELECT lastname, l.street_address
  FROM employees JOIN locations l USING (city);
```

  - With **USING**, the table name or alias cannot be used to qualify the column name, as the join columns are rolled together into a single column.

When you join two tables, the result set consists of all of the rows that matched the join condition (and any other conditions). All rows that do not have a corresponding row in the other table are thrown out. This is called an *inner join*.

The following query will return information only for employees with assignments - no other employees from the **employees** table will have a matching **employee_id** in the **assignments** table.

```
SELECT e.id, e.lastname, a.start_date
  FROM employees e JOIN assignments a ON a.employee_id = e.id;
```

Joins often illustrate one-to-many relationships in your database schema. For example, one location might have many departments:

```
SELECT d.name, l.street_address, l.city, l.state
  FROM departments d JOIN locations l ON l.id = d.location_id;
```

CROSS JOIN — A join done without any join conditions. Every row in the first table is matched with every row in the second table. This is also called a *Cartesian product*.

```
SELECT d.name, l.street_address, l.city, l.state
  FROM departments d CROSS JOIN locations l;
```

This cross join will show the combination of every department with every location - this is seldom useful

Note:

The examples in this chapter use the ANSI standard syntax for joins, which you should always use. However, you might see older join syntax, which is still valid SQL, in which the join conditions are mingled with selection predicates in the **WHERE** clause:

```
SELECT d.name, l.street_address, l.city, l.state
  FROM departments d, locations l
 WHERE l.id = d.location_id;
```

Joining Tables

- Though you can join tables using any columns with compatible data types, parent/child relationships are often used in the join criteria.

```
SELECT e.lastname, e.firstname, j.name, e.salary
  FROM employees e JOIN jobs j ON e.job_id = j.id
 WHERE e.id = 1108;
```

  - This relationship is implemented with a foreign key.

- There is no limit to the number of tables that can be joined together.

  - Each table joined into the query will have its own join criteria, matching its data to data from any of the previously listed tables.

```
SELECT e.lastname, j.name, d.name, e.salary
  FROM jobs j JOIN employees e ON j.id = e.job_id
       JOIN departments d ON d.id = e.department_id;
```

- The join condition can also be complex, involving multiple columns.

```
SELECT e.lastname, e.firstname, e.salary, j.name
  FROM employees e JOIN jobs j
       ON e.job_id = j.id AND e.salary = j.maximum_salary;
```

- All fields from a table can be selected.

```
SELECT e.lastname, e.firstname, e.salary, j.*
  FROM employees e JOIN jobs j
       ON e.job_id = j.id AND e.salary = j.maximum_salary;
```


Joining the same tables using different columns will retrieve very different result sets.

This query retrieves the names of the department managers.

```
SELECT e.id, e.lastname, d.name
  FROM employees e JOIN departments d ON e.id = d.manager_id
 ORDER BY e.id;
```

This query retrieves the names and departments of all employees.

```
SELECT e.id, e.lastname, d.name
  FROM employees e JOIN departments d ON e.department_id = d.id
 ORDER BY e.id;
```

You will also need to carefully decide which tables are joined to which other tables in your query. In this first query, the **employees** table is joined with the **departments** table and the **departments** table to the **locations** table. We are retrieving a list of all employees, with their departments' name and location.

```
SELECT e.id, e.lastname, d.name, l.city, l.state
  FROM employees e JOIN departments d ON e.department_id = d.id
                   JOIN locations l ON d.location_id = l.id
 ORDER BY e.id;
```

This next query involves the same three tables, but they are joined together differently. We are joining the **employess** table to the **locations** table and the **locations** table to the **departments** table. We are retrieving a list of employees who live in the same ZIP code as one of our department locations.

```
SELECT e.id, e.lastname, d.name, l.city, l.state
  FROM employees e JOIN locations l ON e.zipcode = l.zipcode
                   JOIN departments d ON d.location_id = l.id
 ORDER BY e.id;
```

These two queries retrieve very different result sets.

SELF JOINS

- A *self join* queries a single table as though it were two separate tables.
  - Self joins are most frequently performed on tables that model hierarchical data, having foreign keys referencing back into the same table.
- To perform a self join just use two different aliases for the same table.

```
SELECT pp.name AS parent, p.name
  FROM projects p JOIN projects pp
                       ON p.parent_project_id = pp.id
 ORDER BY pp.name;
```


Outer Joins

- An *outer join* retrieves all rows from one table, plus the matching rows from the second table.
  - The column values for the non-matched rows (which would have come from the second table) will be NULL.
- Standard join syntax specifies which table, the **LEFT** or **RIGHT**, will have all of its rows returned.

```
SELECT e.lastname, a.id, a.start_date
  FROM employees e LEFT OUTER JOIN assignments a
                   ON e.id = a.employee_id;
```

  - All records from the **LEFT** table will be returned, with **NULL**s in columns where information from the **RIGHT** table was not available.
  - The **OUTER** keyword is optional.
- **FULL [OUTER] JOIN** — All matching rows from the two tables, plus rows from each table that have no matching rows in the other.
- If you join another table to an outer joined table, you will also need to outer join the new table.

```
SELECT e.lastname, a.id, a.start_date, p.name
  FROM employees e LEFT OUTER JOIN assignments a
                             ON e.id = a.employee_id
                   LEFT OUTER JOIN projects p
                             ON a.project_id = p.id;
```

  - Without the second outer join, you will not be able to match against **NULL** values in the result set, and the rows will be lost.

```
SELECT e.lastname, a.id, a.start_date, p.name
  FROM employees e LEFT OUTER JOIN assignments a
                             ON e.id = a.employee_id
                   JOIN projects p
                             ON a.project_id = p.id;
```


When a row in one joined table does not have a matching row in the other table, the row is left out of the result set of an ordinary (**INNER**) join.

```
SELECT e.lastname, a.id, a.start_date
  FROM employees e JOIN assignments a ON e.id = a.employee_id;
```

With an **OUTER** join, when a row in one joined table does not have a match in the other table, the unmatched row remains in the result set. **NULL** values are supplied for the columns that would have come from the other table.

```
SELECT e.lastname, a.id, a.start_date
  FROM employees e LEFT JOIN assignments a ON e.id = a.employee_id;
```

LEFT and **RIGHT** refer to the position of the table name in the **FROM** clause, relative to the **JOIN** keyword. The **LEFT OUTER JOIN** above can just as easily be written as a **RIGHT OUTER JOIN**:

```
SELECT e.lastname, a.id, a.start_date
  FROM assignments a RIGHT JOIN employees e ON e.id = a.employee_id;
```

EQUIJOINS, NON-EQUIJOINS, AND ANTIJOINS

- Most joins are *equijoins* - the join columns are compared for equality.

```
SELECT lastname, firstname, salary, j.minimum_salary, j.name
  FROM employees e JOIN jobs j
                     ON e.job_id = j.id
                    AND e.salary = j.minimum_salary;
```

  - A *non-equijoin* uses a condition other than equality, such as <, >, **BETWEEN**, or **LIKE** to compare the join columns.

```
SELECT lastname, firstname, salary, j.minimum_salary, j.name
  FROM employees e JOIN jobs j
                     ON e.job_id = j.id
                    AND e.salary > j.minimum_salary;
```

```
SELECT e.lastname, j.name
  FROM employees e JOIN jobs j
        ON e.job_id = j.id
       AND e.lastname LIKE concat(substr(j.name,1,1),'%');
```

- An *antijoin* is an outer join with the matched rows omitted.

```
SELECT e.lastname, a.id, a.start_date
  FROM employees e LEFT OUTER JOIN assignments a
                                ON e.id = a.employee_id
 WHERE a.id IS NULL;
```


joins.sql

A

id	animal
2	cat
3	dog
4	frog
5	giraffe

B

id	sound
1	tweet
2	meow
3	woof
4	ribbet

Cartesian Join:

```
SELECT a.id, a.animal, b.id, b.sound
  FROM a CROSS JOIN b;

SELECT a.id, a.animal, b.id, b.sound
  FROM a, b;
```

a.id	animal	b.id	sound
2	cat	1	tweet
2	cat	2	meow
2	cat	3	woof
2	cat	4	ribbet
3	dog	1	tweet
3	dog	2	meow
3	dog	3	woof
3	dog	4	ribbet
4	frog	1	tweet
4	frog	2	meow
4	frog	3	woof
4	frog	4	ribbet
5	giraffe	1	tweet
5	giraffe	2	meow
5	giraffe	3	woof
5	giraffe	4	ribbet

Inner Join:

```
SELECT a.id, a.animal, b.id, b.sound
  FROM a INNER JOIN b ON a.id = b.id;

SELECT a.id, a.animal, b.id, b.sound
  FROM a INNER JOIN b USING (id);

SELECT a.id, a.animal, b.id, b.sound
  FROM a NATURAL JOIN b;

SELECT a.id, a.animal, b.id, b.sound
  FROM a, b
 WHERE a.id = b.id;
```

a.id	animal	b.id	sound
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet

Left Outer Join:

```
SELECT a.id, a.animal, b.id, b.sound
  FROM a LEFT OUTER JOIN b ON a.id = b.id;

SELECT a.id, a.animal, b.id, b.sound
  FROM a, b
 WHERE a.id = b.id(+);
```

a.id	animal	b.id	sound
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet
5	giraffe	*null*	*null*

Right Outer Join:

```
SELECT a.id, a.animal, b.id, b.sound
  FROM a RIGHT OUTER JOIN b ON a.id = b.id;

SELECT a.id, a.animal, b.id, b.sound
  FROM a, b
 WHERE a.id(+) = b.id;
```

a.id	animal	b.id	sound
null	*null*	1	tweet
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet

Full Outer Join:

```
SELECT a.id, a.animal, b.id, b.sound
  FROM a FULL OUTER JOIN b ON a.id = b.id;

SELECT a.id, a.animal, b.id, b.sound
  FROM a, b WHERE a.id = b.id(+)
 UNION
SELECT a.id, a.animal, b.id, b.sound
  FROM a, b WHERE a.id(+) = b.id;
```

Full Outer Join

a.id	animal	b.id	sound
null	*null*	1	tweet
2	cat	2	meow
3	dog	3	woof
4	frog	4	ribbet
5	giraffe	*null*	*null*

Antijoin:

```
SELECT a.id, a.animal, b.id, b.sound
  FROM a LEFT OUTER JOIN b ON a.id = b.id
 WHERE b.id IS NULL;

SELECT id, animal FROM a
 WHERE id NOT IN (SELECT id FROM b);

SELECT id, animal FROM a
 WHERE NOT EXISTS (SELECT 1 FROM b WHERE b.id = a.id);
```

Antijoin:

a.id	animal	b.id	sound
5	giraffe	*null*	*null*

a.id	animal
5	giraffe

LABS

❶ Using joins, write queries to:

a. List all departments and their manager's first and last name.

b. Find the name of the person who manages the Research department.

c. Find the names and departments of all employees whose department is located in Atlanta.

d. List the names of every employee living in North Carolina with an assignment.

❷ Using joins, write queries to:

a. List the full name of each employee, with their department name and location.

b. List the full name and job of each employee, with their department name and location.

❸ Write a query to list employee assignments. It should retrieve employee id and full name, the assignment start date, and the project start date and name.

❹ Modify the query from ❸ to include the name and start date of the parent project.

❺ Modify the query from ❹ to also list all employees, even those without assignments.

❻ Modify the query from ❺ so that:

* Each employee's department name is included.
* The results are sorted by department and project names.
* Employee name is returned as a single "Firstname Lastname" string.
* Column headers are unambiguous.

Data Manipulation and Transactions

Objectives

- Add new rows to database tables.
- Modify rows in tables.
- Delete rows from tables.
- Perform tasks consisting of more than one SQL statement.
- Make your changes permanent and visible to other users.
- Undo the effects of SQL statements.
- Access database tables concurrently with other users.
- Prevent other users from interfering with your changes.

THE INSERT STATEMENT

- Use the **INSERT** statement to add new rows of data to a table.

```
INSERT INTO table_name [(column_name_list)]
  {VALUES (value_list) | subquery };
```

  - Use ***column_name_list*** to specify the columns for which you will be providing values.
  - You must own the table or have **INSERT** privilege on the table.

- To add a single row to the table, use the **VALUES** clause.

```
INSERT INTO projects (id, name, start_date)
             VALUES (14, 'Project X', CURDATE());
```

  - The order and number of values must match the column list.

- To add zero or more rows at once by copying from existing data, use a subquery.

```
INSERT INTO assignments (project_id,employee_id,start_date)
      SELECT 14, e.id, CURDATE()
        FROM employees e JOIN departments d
                           ON e.department_id = d.id
       WHERE d.name = 'Research';
```

  - The subquery's **SELECT** list can include column values, expressions, and literals.
    - You can generate rows in which some column values come from existing records, while others are provided by the statement itself.

- Use the **NULL** or **DEFAULT** keywords in place of a value to insert a null (if the column permits nulls), or allow a column to be populated with its default value.
  - Most databases support an integer column of type IDENTITY, SERIAL, AUTOINCREMENT etc. for generating primary key values automatically.

```
INSERT INTO locations (street_address, city, state)
              VALUES (NULL, 'Roswell', 'NM');
```


You can enter a new project into the **projects** table with the following:

```
INSERT INTO projects
     VALUES (15, 'Project Y', ADDDATE(CURDATE(),7), NULL, NULL);
```

This format of the **INSERT** command requires that the number and exact order of all columns in the **projects** table be known, and match the VALUES list in correct order. If the table definition should change, this statement would become invalid. A more reliable (and maintainable) format is to specify the order and names of the columns that are being populated:

```
INSERT INTO projects (id, name, start_date, end_date, parent_project_id)
     VALUES (15, 'Project Y', ADDDATE(CURDATE(),7), NULL, NULL);
```

This format is required when populating only a subset of all columns in a table. Any skipped columns will be automatically assigned the specified default or (if allowed) null value. The **INSERT** below is equivalent to those above, unless either of the last two columns has a non-NULL **DEFAULT** value:

```
INSERT INTO projects (id, name, start_date)
     VALUES (15, 'Project Y', ADDDATE(CURDATE(),7));
```

Databases vary as to whether you can provide an explicit value for an automatically incremented primary key column; if you can, your value must not duplicate an existing one (which is true for any primary key column.) You can always leave such a column out of the **INSERT** statement:

```
INSERT INTO projects (name, start_date)
     VALUES ('Project Y', ADDDATE(CURDATE(),7));
```

A subquery can be used to easily populate rows from other data in the database. To add another employee to the database with the same address and employee information as an existing employee:

```
INSERT INTO employees (firstname, lastname, extension, hiredate, salary,
                       department_id, job_id, address, city, state, zipcode)
                  SELECT 'John', 'Walker', extension, CURDATE(), salary,
                       department_id, job_id, address, city, state, zipcode
                    FROM employees
                   WHERE id=1133;
```

THE UPDATE STATEMENT

- Use the **UPDATE** statement to change existing data in a single table.

```
UPDATE table_name
  SET column=expression [,column=expression,...]
[WHERE condition];
```

  - You must either own the table or have **UPDATE** privilege.

```
UPDATE jobs
   SET minimum_salary = 42000,
       maximum_salary = 65000
 WHERE id = 19;
```

- The ***expression*** may be a subquery that must return only one row.

```
UPDATE employees
   SET department_id = (SELECT id
                          FROM departments
                         WHERE manager_id = 1062)
 WHERE id = 1132;
```

  - Specify **DEFAULT** to set a value back to the default value specified for this column when the table was created.

```
UPDATE employees
   SET extension = DEFAULT
 WHERE id = 1240;
```

    - A **NULL** will be issued if there is no default for this column.

- The **WHERE** clause identifies the rows to be updated.

  - If no **WHERE** clause is used, all rows in ***table_name*** are updated.

With a correlated subquery, we can update each employee based on their department.

```
UPDATE  employees
   SET  salary = (SELECT maximum_salary
                    FROM jobs
                   WHERE id = employees.job_id)
 WHERE department_id = 1;
```

In this example, however, the subquery must be executed once for every row processed by the **UPDATE**, since the value returned by the **SELECT** will depend on the value of the **employees.job_id** in the current row being updated.

Always study your database vendor's documentation to see what **UPDATE** statement variations are supported.

THE DELETE STATEMENT

- The **DELETE** statement removes rows from a table.
- You must either own the table or have **DELETE** privilege.
- The format of the **DELETE** command is:

```
DELETE FROM table_name
[WHERE condition];
```

- The **WHERE** clause identifies the rows to be deleted.

delete.sql

```
DELETE FROM assignments
 WHERE project_id = 14
   AND employee_id = 1049;
```

  - The ***condition*** can be the same as those found in the **SELECT** statement.
  - If no **WHERE** clause is used, all rows in ***table_name*** are deleted!

In addition, many databases provide the **TRUNCATE** statement. **TRUNCATE** deletes all of the rows in a table. Unlike the **DELETE** statement, you cannot specify any conditions on which rows are to be deleted. The example below will delete the contents of the **assignments** table:

```
TRUNCATE TABLE assignments;
```

TRUNCATE is used instead of the **DELETE** statement to free space allocated for the table; and, because each deletion is not logged, the **TRUNCATE** statement is usually faster than **DELETE**. The **TRUNCATE** statement cannot be undone (see **ROLLBACK**) and should, therefore, be used with caution.

TRANSACTION MANAGEMENT

- A database *transaction* groups a series of DML statements into a single logical unit of work.

- At any point in a transaction, you can roll back all of your DML statements.

```
ROLLBACK;
```

  - All of your changes are discarded and a new transaction may begin.

- None of your changes are visible in the database until you commit your transaction.

```
COMMIT;
```

  - All your changes are made permanent and a new transaction may begin.

- **COMMIT** as soon as you have successfully completed the statements making up a logical unit of work.

  - Transactions should have tightly-limited scope.

- Some databases operate by default in *autocommit* mode - each individual **INSERT**, **UPDATE**, or **DELETE** is immediately committed and can't be rolled back.

  - Explicitly begin a multistatement transaction with the **SET TRANSACTION** or **START TRANSACTION** command.

  - You can add savepoints to your transactions to give yourself a place to partially rollback to.

```
SAVEPOINT step1;
```

```
ROLLBACK TO step1;
```


Transactions help you manage tasks that must succeed as a logical group. To change an employee's id from 1120 to 2222, for example, you must also change any corresponding child records:

1. Start a transaction:

```
START TRANSACTION;
```

2. Insert a new employee record:

```
INSERT INTO employees (id, firstname, middlename, lastname, gender,
                       email, extension, hiredate, salary,
                       commission_pct, department_id, job_id, address,
                       city, state, zipcode, version)
                SELECT 2222, firstname, middlename, lastname, gender,
                       email, extension, hiredate, salary,
                       commission_pct, department_id, job_id, address,
                       city, state, zipcode, version
                  FROM employees
                 WHERE id=1120;
```

2. Update any child records to refer to the new **id**:

```
UPDATE assignments SET employee_id = 2222 WHERE employee_id = 1120;
UPDATE departments SET manager_id = 2222 WHERE manager_id = 1120;
```

3. Delete the old employee record:

```
DELETE FROM employees WHERE id = 1120;
```

4. If all three statements succeed, then commit the work:

```
COMMIT;
```

However, if any of the **UPDATE** or **DELETE** statements fail, and we haven't committed yet, then we can roll the work back:

```
ROLLBACK;
```

Note that if we had committed, then the **ROLLBACK** statement would have no effect.

Since the RDBMS must temporarily store enough information to either commit or roll back your entire transaction, it could run out of space if you do not commit often enough. A **DELETE** statement that deletes 100,000 rows from a table may cause the RDBMS to make temporary copies of all those rows. If the system does not have sufficient space for this, the delete will fail. You might have to delete those rows in smaller chunks, committing each time.

Concurrency

- Databases use *locks* of various types to coordinate data manipulation, and many other activities, among concurrent sessions.

- When concurrent sessions access the same rows and perform DML statements — **INSERT**s, **UPDATE**s, and **DELETE**s — the database isolates each session's work.

  - This assures one session can't update or delete a row another session has already updated, until the other session does a **COMMIT** (or **ROLLBACK**).

- When a transaction begins manipulating data, it creates an *Exclusive Lock*, and associates all inserted, modified, or deleted rows with that lock.

  - Another transaction trying to manipulate any of the rows will find the lock and wait.

  - When the first transaction commits or rolls back, its lock clears, and the next transaction in the queue can lock the rows it needs.

- A deadlock can occur if one transaction requires access to data locked by another in order to complete, while at the same time holding a lock that the other transaction is waiting for.

  - Most databases automatically detect this situation, rolling back statements or transactions to clear the deadlock.

- It's possible to explicitly lock data without performing DML:

  - **LOCK TABLES *tablename* WRITE;**

  - **SELECT * FROM *tablename* FOR UPDATE;**

Locking is one of the more intricate and complicated features of a database, and the internal details vary from vendor to vendor.

LOADING TABLES FROM EXTERNAL SOURCES

- Many systems provide statements or utilities that allow you to load bulk data from external files.

- The MySQL **LOAD DATA** command loads a file into a table according to the format and other options specified.

  - The **INFILE** parameter specifies the filename.

    - By default this refers to a file on the MySQL server host, with the path relative to MySQL's data directory.

    - Use **LOCAL INFILE** to specify a file relative to the location from which you're running the **mysql** client.

- The **mysqlimport** utility loads rows into a table from external data files.

  - The name of the datafile determines which table the data will load.

  - Command-line options specify the file's format, following the syntax of **LOAD DATA**.

- The **mysqldump** utility exports a database's table definitions and/or data, which you can redirect to a file.

```
mysqldump mydb > mydbdump.sql
```

  - The dump contains standard SQL commands to create and insert data into the dumped tables; you can run this as a script using **mysql**.

```
mysql -u someuser -p < mydbdump.sql
```


Oracle provides a standalone utility, SQL*Loader, for loading rows into a table from an external source. SQL*Loader can read a wide variety of data files, parsing records (according to your specifications) into values of the correct types for your table's columns.

empdata.txt

```
LNAME,FNAME,DNUM,PAYAMT,JOBNUM
"Tsang","Leila",4,20000,26
"Sierra","Lisa",3,30000,22
"Happ","Bill",4,18000,26
"Garia","Pete",3,31000,22
"Ehrlich","Donna",3,30000,22
"Clemens","Edith",5,25000,8,23
"Caner","Michael",3,30000,22
"Gonzales","Pamela",6,25000,23
"Merlick","Mary",2,30000,22
"Hausuar","Keith",2,30000,22
"Plank","Scott",2,32000,22
"Minor","Sammy",4,18000,26
```

Use LOAD DATA to parse and load this into the **employees** table:

```
LOAD DATA
LOCAL INFILE 'empdata.txt'
IGNORE
INTO TABLE employees
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
IGNORE 1 LINES
(lastname, firstname, department_id, salary, job_id, hiredate)
SET hiredate = CURRENT_DATE;
```

LABS

Throught these labs, be sure to consider whether to start a transaction so you can roll back a change.

❶ Insert your own name and address into the **employees** table choosing an appropriate department and job ID.

❷ Set your salary to be halfway between the min and max salaries for your job.

❸ With a single statement, set your city, state, and zipcode to match your department's.

❹ Assign yourself to the "Flux Capacitor" project.

❺ Congratulations — for doing all this, you get a 5% raise.

❻ Delete all the employee project assignments.

❼ What — are you nuts!? Get those assignments back!

Aggregate Functions and Advanced Techniques

Objectives

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

Subqueries

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

```
SELECT lastname, firstname, city, zipcode
  FROM employees
 WHERE id = (SELECT manager_id
               FROM departments
              WHERE name = 'HR');
```

  - Operators such as **IN** allow a multi-row subquery, one that can return any number of rows.

```
SELECT lastname, firstname, city, state
  FROM employees
 WHERE zipcode IN (SELECT zipcode FROM locations);
```


Show the manager of department #4:

```
SELECT firstname, lastname
  FROM employees
 WHERE id = (SELECT manager_id
               FROM departments
              WHERE id = 4);
```

Show the name of the employee of assignment #17:

```
SELECT lastname, firstname
  FROM employees
 WHERE id = (SELECT employee_id
               FROM assignments
              WHERE id = 17);
```

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

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

```
SELECT lastname, firstname
  FROM employees
 WHERE id IN (SELECT employee_id FROM assignments);
```

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

```
SELECT lastname, firstname, start_date
  FROM employees JOIN assignments
                   ON employees.id = assignments.employee_id;
```

Correlated Subqueries

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

```
SELECT p.name, p.start_date
  FROM projects p JOIN assignments a ON p.id = a.project_id
 WHERE a.employee_id = (SELECT id
                          FROM employees
                         WHERE id = a.employee_id
                           AND hiredate < p.end_date);
```

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

```
SELECT p.name, p.start_date
  FROM projects p JOIN assignments a ON p.id = a.project_id
                  JOIN employees e ON a.employee_id = e.id
 WHERE e.hiredate < p.end_date;
```

  - However, subqueries can help you factor a very complex query into manageable pieces.

The EXISTS Operator

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

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

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

```
SELECT e.firstname, e.lastname
  FROM employees e
 WHERE EXISTS (SELECT *
                 FROM assignments
                WHERE employee_id = e.id);
```

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

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

```
SELECT e.firstname, e.lastname
  FROM employees e
 WHERE NOT EXISTS (SELECT *
                     FROM assignments
                    WHERE employee_id = e.id);
```

```
SELECT e.firstname, e.lastname
  FROM employees e
 WHERE id NOT IN (SELECT employee_id
                    FROM assignments);
```

```
SELECT e.firstname, e.lastname
  FROM employees e LEFT OUTER JOIN (SELECT employee_id
                                      FROM assignments) a
                                   ON e.id = a.employee_id
 WHERE a.employee_id IS NULL;
```

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

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

```
SELECT e.firstname, e.lastname
  FROM employees e JOIN assignments a
                     ON e.id != a.employee_id;
```

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

The Aggregate Functions

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

```
SELECT SUM(salary * 1.5) AS tce
  FROM employees
 WHERE department_id = 1;
```

- Aggregate functions are sometimes called *grouping*, *column*, or *set* functions.
- Aggregate functions cannot be used in a **WHERE** clause.
  - However, you can place the aggregate in a subquery's **SELECT** list.

```
SELECT id, firstname, lastname, salary
  FROM employees e
 WHERE salary < (SELECT MAX(salary)/4
                   FROM employees);
```



```
SELECT COUNT(*) FROM employees;

SELECT AVG(salary) FROM employees;

SELECT MIN(salary) FROM employees
 WHERE department_id = 3;

SELECT MAX(invoice_number) FROM order_header;
```

Multiple aggregates can be used in a single query.

```
SELECT MIN(salary), AVG(salary), MAX(salary)
  FROM employees
 WHERE department_id = 3;
```

NULLS AND DISTINCT

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

```
SELECT COUNT(end_date)
   FROM assignments;
```

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

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

```
SELECT COUNT(DISTINCT end_date)
 FROM assignments;
```


The following query will return the total number of assignment records.

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

This query will return the number of assignments with known end dates.

```
SELECT  COUNT(end_date)
  FROM assignments;
```

This last query will return the number of unique end dates of assignments.

```
SELECT COUNT(DISTINCT end_date)
  FROM assignments;
```

Grouping Rows

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

```
SELECT d.name, SUM(e.salary)
  FROM employees e JOIN departments d
                       ON e.department_id = d.id
 GROUP BY d.name;
```

- The rows are organized into groups in which the values of the grouping column are equal.
  - The aggregate function is then evaluated once for each group.
  - The **SELECT** statement returns a single summary row for each group.
  - Only aggregate functions and the column(s) used in the **GROUP BY** clause are permitted in the **SELECT** clause.
- **HAVING** can be used to eliminate group summary rows based on the value of their aggregations.

```
SELECT d.name, SUM(e.salary)
  FROM employees e JOIN departments d
                     ON e.department_id = d.id
 GROUP BY d.name
HAVING SUM(salary) >= 500000;
```



```
SELECT j.name, AVG(e.salary) AS average_pay
  FROM employees e JOIN jobs j ON e.job_id = j.id
 GROUP BY j.name;
```

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

```
SELECT j.name, AVG(e.salary) AS average_pay
  FROM employees e JOIN jobs j ON e.job_id = j.id
 WHERE e.state = 'NC'
 GROUP BY j.name;
```

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

```
SELECT d.name, j.name, AVG(e.salary) AS average_pay
  FROM departments d JOIN employees e ON d.id = e.department_id
                     JOIN jobs j ON e.job_id = j.id
 WHERE e.state = 'NC'
 GROUP BY d.name, j.name;
```

The database software can do this kind of analysis much more efficiently, over potentially very large amounts of data, than you would be able to do be retri\ eving the raw data and crunching numbers in a spreadsheet or with program code.

```
SELECT p.name, AVG(e.salary), SUM(e.salary)
  FROM employees e JOIN assignments a ON e.id = a.employee_id
                   JOIN projects p ON a.project_id = p.id
 GROUP BY p.name;
```

```
SELECT l.city, l.state, COUNT(e.id)
  FROM employees e JOIN departments d ON e.department_id = d.id
                   JOIN locations l ON d.location_id = l.id
 GROUP BY l.city, l.state;
```

```
SELECT l.city, l.state, l.square_footage/COUNT(e.id) AS sqft_per_emp
  FROM employees e JOIN departments d ON e.department_id = d.id
                   JOIN locations l ON d.location_id = l.id
 GROUP BY l.city, l.state;
```

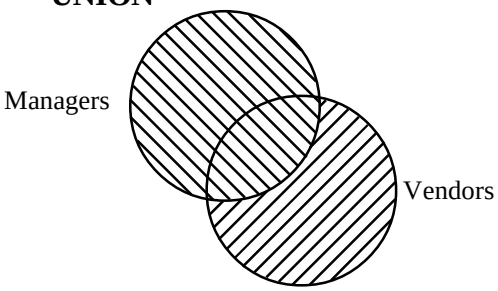
Combining SELECT Statements

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

```
SELECT id, name
   FROM departments
 UNION
SELECT id, name
  FROM jobs
 ORDER BY name;
```

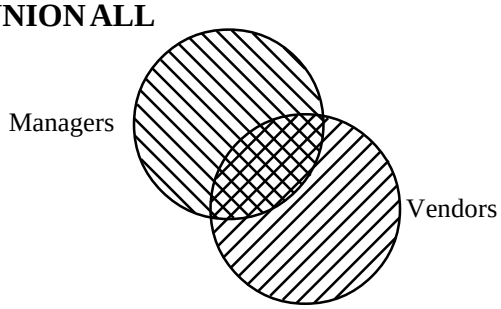
  - The **UNION ALL** operator will retain duplicates.
- Some database products also support the **INTERSECT** (***AND***) operator that returns rows common to both, and the **EXCEPT** or **MINUS** (***AND NOT***) operator returning all rows in the first query not returned by the second (another way of producing an antijoin.)

UNION



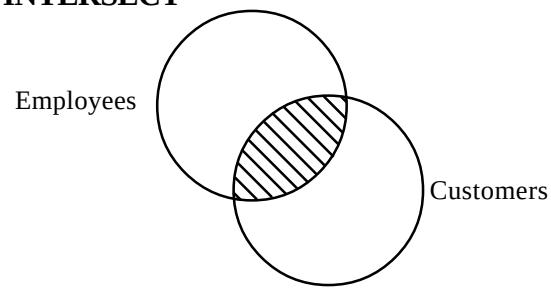

```
SELECT manager_id
  FROM store
 UNION
SELECT vendor_rep_id
  FROM vendor;
```

UNION ALL



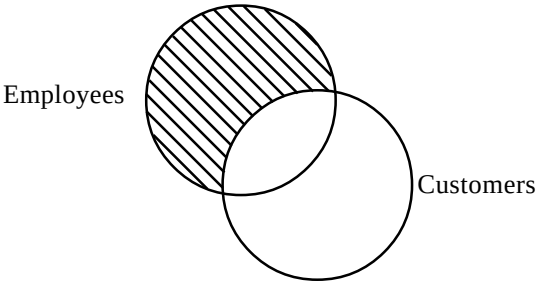

```
   SELECT manager_id
     FROM store
UNION ALL
   SELECT vendor_rep_id
     FROM vendor;
```

INTERSECT




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

MINUS




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

LABS

❶ Write single queries that will:

a. Retrieve the total number of employees of department #6.

b. Retrieve the total number of employees of department #6 who live in Atlanta .

c. Retrieve the total number of completed assignments.

d. Report the average salary for testers.

e. Report the names and salaries of the highest paid employee(s) of the company.

f. Report the names and salaries of the lowest paid employee(s) of the company.

g. Report all employees making less than the average wage.

h. Report the total amount of salaries paid to employees of department #3.

i. Report the total amount of salaries paid to employees of department #3 whose job title includes the word "Developer".

❷ Write single queries that will:

a. Report the number of employees for each department.

b. Report the number of employees and total salaries for each department.

c. Report the number of employees and total salaries for each department that has more than 15 employees.

d. Report the number of employeess and their total salaries by department and project.

e. Report the average employee income based on job title and state.

