SQL

Rodrigo García Carmona

Universidad San Pablo-CEU

Escuela Politécnica Superior

DDL

DATA TYPES

- All columns must have a data type. The most common data types in SQL are:
- **Alphanumeric:**
  - Fixed length: `CHAR(n)`
  - Variable length: `VARCHAR(n)`
  - …or just: `TEXT`
- **Numeric:**
  - `DECIMAL(precision[,scale])` *For instance:* `DECIMAL(10,2)`.
  - `INTEGER (or INT)`
  - `REAL (or FLOAT, DOUBLE)`
- **Time and Date:**
  - `DATE:` YYYY-MM-DD
  - `TIME:` HH:MI:SS
  - `DATETIME (also TIMESTAMP):` YYYY-MM-DD HH:MI:SS
- **Other data types:**
  - `BLOB`
  - `BOOLEAN`

USER-DEFINED DATA TYPES

- **We can define our own custom data types.**
- `CREATE DATATYPE custom_name data_type [ [ NOT ] NULL ] [ DEFAULT default_value ] [ CHECK ( predicate ) ]`
- Can have conditions in the form of CHECK clauses, and default values. We can also indicate if null values are allowed. These conditions are inherited by any column with that data type.
- If a condition is specified over a column using a custom data type, the new condition has preference over the user-defined data type's conditions.
- **Examples:**
  - `CREATE DATATYPE address VARCHAR(50) NULL`
    - "address" data type: string of up to 50 characters that can be NULL.
  - `CREATE DATATYPE id INTEGER NOT NULL CHECK (id >= 0)`
    - "id" data type: positive integer that can't be NULL.

TABLES DEFINITION

- **Before inserting data we must define the relations:**
- ```
  CREATE TABLE table_name (
  column_definition [ [ NOT ] NULL ] [ DEFAULT default_value ]
  [ [ CONSTRAINT name ] column_constraint ],
  ...
  [ [ CONSTRAINT name ] table_constraint ],
  ...)
  ```
- **Column definitions:**
  - `column_name data_type`
  - Defines a column for the table. Only the already defined data types are allowed. Two or more columns of the same table can't have the same name.
  - If NOT NULL is specified, or if a PRIMARY KEY restriction is applied, the column can't accept null values.
  - **Possible default values:**
    - `character_string, number`
    - `CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP`
    - `NULL`
  ```

TABLE CONSTRAINTS

- **The following constraints can be applied to a table:**
- `UNIQUE ( column_name, ... )`
  - The column (or columns) univocally identifies each record.
- `PRIMARY KEY ( column_name, ... )`
  - The same as UNIQUE + NOT NULL, but a table can only have one primary key.
- `CHECK ( predicate )`
  - Impose conditions that the column's values must comply to. For instance, this feature could be used to restrict the type of a “gender” column to “male” or “female”. Predicates are explained later in this unit.
- `FOREIGN KEY [ alias ] [ ( column_name, ... ) ]`
  `REFERENCES table_name [ ( column_name, ... ) ]`
  `[ actions ]`
  - Constraints the values of a column set to be the same as the values for another table's primary key and/or unique columns. This constraint is called a **foreign key**.
  - In most cases, a foreign key references a primary key.

COLUMN CONSTRAINTS

- Very similar to table constraints. In fact, it's another way of conveying the same information.
- If the constraint only affects one column it's usually better to use column constraints instead of table constraints.
- **The following constraints can be applied to a column:**
  - `UNIQUE`
  - `PRIMARY KEY`
    - In SQLite, a PRIMARY KEY should be an INTEGER.
  - `REFERENCES table_name [ ( column_name ) ] [ actions ]`
    - Note that with column constraints we don't need to write FOREIGN KEY.
  - `CHECK ( predicate )`
- These two sentences produce the same result:
  - `CREATE TABLE town (code INTEGER UNIQUE);`
  - `CREATE TABLE town (code INTEGER, UNIQUE (code));`

AUTOINCREMENT

- **AUTOINCREMENT** is a special constraint, that can only be used in certain SQL solutions. Sometimes it has a different meaning for each solution.
- SQLite supports AUTOINCREMENT only for columns that:
  - Are of the INTEGER type.
  - Have the PRIMARY KEY constraint.
- During an INSERT, if the PRIMARY KEY column is not explicitly given a value, then it will be filled automatically with an unused integer, usually one more than the largest currently in use.
- AUTOINCREMENT prevents the reuse of values over the **lifetime** of the database. In other words, it prevents the reuse of values from previously deleted rows.

ACTIONS

- Actions are used to maintain the relationships with foreign keys. When a primary key is modified or deleted in a table, the values corresponding to the foreign keys in other tables must also be modified.
- To activate them in SQLite we must run the following sentence every time a connection is established:
  - **PRAGMA foreign_keys = ON**
- **Actions:**
  - `[ ON UPDATE action_type ] [ ON DELETE action_type ]`
- **Action types:**
  - CASCADE
    - Used with ON UPDATE: updates the foreign keys to the new primary key's value.
    - Used with ON DELETE: deletes the records whose foreign keys are the same as the deleted record's primary keys.
  - SET NULL
    - Sets all foreign keys that correspond to the updated or deleted primary key to null.
  - SET DEFAULT
    - Sets all foreign keys that correspond to the updated or deleted primary key to the value specified in the DEFAULT clause.
  - RESTRICT
    - Prevents the update or deletion of a primary key if there are foreign keys referencing it.

SAMPLE TABLE CREATIONS

- Creates a table with columns **dep_id** (primary key), **name** and **location**, allowing for null values in name and location:
  - ```
    CREATE TABLE departments (
    dep_id INTEGER PRIMARY KEY,
    name TEXT,
    location TEXT
    );
    ```
- Creates an employee table that references the departments' primary key with a foreign key, and takes the SET NULL actions for deletions and the CASCADE actions for updates:
  - ```
    CREATE TABLE employees (
    emp_id INTEGER,
    name TEXT UNIQUE,
    manager TEXT UNIQUE,
    salary REAL,
    bonus REAL,
    job TEXT,
    hiredate DATE,
    dep_id INTeger,
    PRIMARY KEY (emp_id),
    FOREIGN KEY (dep_id) REFERENCES departments(dep_id)
                         ON DELETE SET NULL ON UPDATE CASCADE);
    ```

INDEXES

- **Creating an index:**
  - `CREATE [ UNIQUE ] INDEX index_name`
    `ON table_name ( column_name [ ASC | DESC ], ... )`
- Indexes are special lookup tables that the DBMS uses to speed up data retrieval. They're similar to an index in the back of a book. Indexes are automatically used by the DBMS. An index speeds up SELECTs, but slows down UPDATEs and INSERTs.
- Once created, an index can only be referenced to delete it.
- The UNIQUE constraint forbids several records from having the same value for the index attribute.
- Indexes are ascending by default, but descending indexes can be created using DESC.
- The DBMS automatically creates indexes for the primary keys and the columns with UNIQUE restrictions.
- **Example:**
  - `CREATE UNIQUE INDEX ix_emp ON employees (manager);`

MODIFYING A TABLE'S STRUCTURE

- We can modify an already created table's structure.
- **Modifying a table:**
- ```
  ALTER TABLE table_name
  { [ ADD | MODIFY ] column_definition [ [ NOT ] NULL ]
  [ DEFAULT default_value ] [ [ CONSTRAINT name ] column_constraint ] |
  ADD [ CONSTRAINT name ] table_constraint |
  RENAME COLUMN column_name TO new_name |
  DROP COLUMN column_name | RENAME TO new_table_name }
  ```
- **Examples:**
  - Add a new column for surnames to the employees table:
    - ```
      ALTER TABLE employees
      ADD surname TEXT;
      ```
  - Add a new restriction to employee table. The manager must be another employee:
    - ```
      ALTER TABLE employees
      ADD CONSTRAINT mgr_const FOREIGN KEY (manager) REFERENCES
      employees(name);
      ```
- SQLite **only supports** the RENAME TO and ADD COLUMN variants of the ALTER TABLE command.
      ```

DELETING OBJECTS

- We can delete already created objects.
- **Usage:**
  - `DROP [ DATATYPE | INDEX | CONSTRAINT | TABLE ] object_name`
- **Examples:**
  - Datatype deletion:
    - `DROP DATATYPE address;`
  - Index deletion:
    - `DROP INDEX employees.ix_emp;`
  - Constraint deletion (if it has a name!):
    - `DROP CONSTRAINT employees.const_emp;`
  - Table deletion:
    - `DROP TABLE employees;`

DML

QUERY

- **Query for results in a database:**
- ```
  SELECT [ DISTINCT ] { expr_1 [ , expr_2 ] ... | * }
  FROM table_name
  [ WHERE predicate ]
  [ ORDER BY column1 [ , column2 ]... [ ASC | DESC ] ]
  ```
- Expressions (usually column names) are written after the SELECT clause. “*” means all columns form the table.
- The result of a query is always a table. This table's columns are those appearing in the expressions, in the order they are listed.
- If DISTINCT is used, duplicate results will be removed.
- Only results complying with the WHERE condition are shown.
- Results can be ordered with ORDER BY, ascending (default) or descending. If no order is specified the results won't be ordered. Instead of the column name we can use the position in which the column appears in the expression.
  ```

SAMPLE QUERIES

- List all employees with all its associated data.
  - `SELECT * FROM employees;`
- List all employees, showing only their names and managers:
  - `SELECT name, manager FROM employees;`
- List all employees, showing only their names, and order them by manager:
  - `SELECT name FROM employees ORDER BY manager;`
- List all employees, showing their names, managers and salaries, and order them by manager and then by department, descending:
  - `SELECT name, manager, dep_id FROM employees ORDER BY 2, 3 DESC;`
- List all different managers:
  - `SELECT DISTINCT manager FROM employees;`

EXPRESSIONS

- Expressions are formulas that can be used to perform operations over the data while making a SELECT query. For instance, we could obtain the product of two columns or the division of a column by a constant.
- Expressions can also be used as part of predicates, in the search conditions featured in the WHERE clause.
- An expression is a combination of **operators**, **operands** and **parenthesis**. The result of an expression's execution is a single value.
- **Operands** can be column names, constants or other expressions.
- **Operators over numerical types:**
  - Addition: +
  - Subtraction: -
  - Product: *
  - Division: /
- Operators over alphanumeric data heavily depend on the particular SQL implementation.

SAMPLE QUERIES WITH EXPRESSIONS

- An expression can be followed by a string, which will be displayed as the column's name for that expression's result.
- **Examples:**
  - Get the name and the annual salary for each employee and display it as “Annual Salary” :
    - ```
      SELECT name, salary * 4 * 12 'Annual Salary'
      FROM employees;
      ```
  - Show the employee's name and a column that contains his or her salary with the bonus. The new salary must be displayed as “Salary with Bonus” :
    - ```
      SELECT name, salary + bonus 'Salary with Bonus'
      FROM employees;
      ```

PREDICATES

- Up until this moment we have seen basic queries, but we can specify more elaborate search conditions.
- A **predicate** models a logical statement, is performed over column values, and can return “true”, “false” or “unknown”.
- Predicates are used with:
  - CHECK clauses for the DDL.
  - WHERE clauses for the DML.
- A predicate is satisfied only if its value is “true”. Columns that return “false” and “unknown” are rejected.
- **Basic predicates:** Comparison between two values.
  - x = y : “true” if x equals y.
  - x <> y : “true” if x is not equal to y.
  - x < y : “true” if x is lesser than y.
  - x > y : “true” if x is bigger than y.
  - x <= y : “true” if x is lesser than or equal to y.
  - x >= y : “true” if x is bigger than or equal to y.

SAMPLE PREDICATES

- List all employees that work as salesmen:
  - ```
    SELECT name
    FROM employees
    WHERE job = 'salesman';
    ```
- List all employees that don't work at department 30:
  - ```
    SELECT name
    FROM employees
    WHERE dep_id <> 30;
    ```
- List all employees hired before January 1st 2010:
  - ```
    SELECT name
    FROM employee
    WHERE hiredate < '2010-1-1';
    ```
- List the name and hire date of all analysts:
  - ```
    SELECT name, hiredate
    FROM employees
    WHERE job = 'analyst';
    ```

SUBORDINATE QUERIES

- The second operand of a predicate can be the result of an execution of another SELECT query. This operand is then called a **subordinate query**.
- A subordinate query must be within parenthesis and return just one value. That is, the table resulting from the subordinate SELECT must have exactly one column and up to one row.
- If the subordinate SELECT's result is an empty table, its value is considered to be “unknown”.
- A subordinate query can't use the ORDER BY clause.

SAMPLE SUBORDINATE QUERIES

- Find all employees whose salary is better than Adams:
  - ```
    SELECT name FROM employees
    WHERE salary > (SELECT salary FROM employees
                    WHERE name = 'Adams');
    ```
- Find all employees that work in the same department as Clark:
  - ```
    SELECT name FROM employees
    WHERE dep_id = (SELECT dep_id FROM employees
                    WHERE name = 'Clark');
    ```
- Find all employees whose manager earns more than 1000:
  - ```
    SELECT name FROM employees
    WHERE manager = (SELECT name FROM employees
                     WHERE salary > 1000);
    ```
    ```

COMPOUND PREDICATES

- **Compound predicates** are combinations of predicates made using the AND, OR and NOT logical operators.
- AND and OR are used with two predicates, while NOT is only used with one.
- Parenthesis can be used.
- **Examples:**
- Find the name of all salesmen that earn more than 1500:
  - ```
    SELECT name FROM employees
    WHERE job = 'salesman' AND salary > 1500;
    ```
- Find the names of all employees that work as clerks or work in department 30:
  - ```
    SELECT name FROM employees
    WHERE job = 'clerk' OR dep_id = 30;
    ```

PREDICATES AND QUANTIFIERS

- **Universal quantifier (for all):**
  - **ALL:** Returns true if the condition is true for all values returned by the subordinate SELECT clause.
  - **Example:** Find the names of all employees that earn more than all the salesmen:
    - ```
      SELECT name FROM employees
      WHERE salary > ALL (SELECT salary FROM employees
                          WHERE job = 'salesman');
      ```
- **Existential quantifier (exists):**
  - **SOME / ANY:** Returns true if the condition is true for at least one of the values returned by the subordinate SELECT clause.
  - SOME and ANY are interchangeable.
  - **Example:** Find the names of all employees that earn more than at least one of the salesmen:
    - ```
      SELECT name FROM employees
      WHERE salary > SOME (SELECT salary FROM employees
                           WHERE job = 'salesman');
      ```

TESTS FOR PREDICATES

- **Null value test:**
  - `column_name IS [ NOT ] NULL`
  - Filters the records with a value of null (or different to null) from the result.
  - **Example:** Finds the employees that have a bonus:
    - `SELECT name FROM employees WHERE bonus IS NOT NULL;`
- **Belongs to a set test:**
  - `predicate [ NOT ] IN ( value_1 [ , value_2 ] ... )`
  - Finds if the result predicate is in the set of values specified after IN.
  - **Example:** Find all employees named “Julius”, “Allen” or “Scott”:
    - ```
      SELECT * FROM employees
      WHERE name IN ('Julius', 'Allen', 'Scott');
      ```
      ```

COMPARING PREDICATES WITH VALUES

- **BETWEEN - AND:**
  - `predicate_1 [ NOT ] BETWEEN predicate_2 AND predicate_3`
  - To find if a value is between two other values (inclusive).
  - **Example:** Find all employees that earn between 2000 and 3000:
    - `SELECT name FROM employees WHERE salary BETWEEN 2000 AND 3000;`
- **LIKE:**
  - `column_name [ NOT ] LIKE pattern`
  - To find values that comply with the provided alphanumerical pattern.
  - The pattern can use any valid alphanumerical character.
  - In the pattern "_" means “any character, but just one”; "%" means “any combination of characters, whatever the length” (even empty strings); “[charlist]” means any character in the list; “[!charlist]” means any character not in the list.
  - **Example:** Find all employees whose name has a “d” as the second character:
    - `SELECT name FROM employees WHERE name LIKE '_d%';`

COLUMN FUNCTIONS

- Return a single value after executing an operation over all values of one or more columns.
- The argument of this operation is a collection of values taken from one or more columns.
- Column functions:
  - **AVG** returns the average value of the collection.
  - **MAX** returns the highest value in the collection.
  - **MIN** returns the smallest value in the collection.
  - **SUM** returns the sum of all values in the collection.
  - **COUNT** returns the number of elements in the collection.
- The argument is usually an expression. Column functions can also be used as operands in the predicates used in WHERE clauses, but only in subordinate queries.
- Before executing these functions, records can be split in several groups, using the GROUP BY clause. Column functions are executed over all groups separately.
- Before executing these functions, undesired records can be removed, using the HAVING clause.

SAMPLE COLUMN FUNCTIONS

- Find how many employees have a bonus and the average bonus:
  - `SELECT COUNT(bonus), AVG(bonus) FROM employees;`
- Find the average salary of analysts:
  - ```
    SELECT AVG(salary) 'Average Salary' FROM employees
    WHERE job = 'analyst';
    ```
- Find the lowest and highest salary of all analysts, and the difference between them:
  - ```
    SELECT MAX(salary), MIN(salary), MAX(salary)-MIN(salary)
    FROM employees WHERE job = 'analyst';
    ```
- Find how many different job types are in department 30:
  - `SELECT COUNT(DISTINCT job) FROM employees WHERE dep_id = 30;`
- Find how much money the company pays every year:
  - `SELECT SUM(salary * 12) 'Expenses' FROM employees;`
- Find how many employees in department 30 are earning more than the average salesman:
  - ```
    SELECT COUNT(name) FROM employees
    WHERE dep_id = 30 AND salary > (SELECT AVG(salary) FROM employees
                                    WHERE job = 'salesman');
    ```
    ```

ROW GROUPING FOR COLUMN FUNCTIONS

- We can group several rows together before executing a column function. This way, the function will be executed over each group separately.
- **GROUP BY:** This clause can be used in SELECT operations. It must appear after the WHERE clause, if it exists.
- **Usage:**
  - `GROUP BY column_name_1 [ , column_name_2] ...`
- **Examples:**
- Find the lowest, highest and average salary of all employees, grouped by job:
  - ```
    SELECT job, MIN(salary), MAX(salary) FROM employees
    GROUP BY job;
    ```
- Find how many employees of each department have a higher than average salary:
  - ```
    SELECT dep_id, COUNT(name) FROM employees
    WHERE salary > (SELECT AVG(salary) FROM employees)
    GROUP BY dep_id;
    ```

ROW DISCARDING FOR COLUMN FUNCTIONS

- We can also remove unwanted rows for column functions.
- **HAVING:** This clause can be used in SELECT operations, together with GROUP BY. Removes the rows that do not fulfill a specified criteria **after** doing the grouping. WHERE works over the ungrouped rows, **before** doing the grouping.
- **Usage:**
  - `HAVING expression`
  - The expression can contain functions. WHERE expressions can't.
- **Examples:**
- Find the lowest and average salaries for each position. Consider only the positions with an average salary higher than 1500:
  - ```
    SELECT job, AVG(salary), MIN(salary) FROM employees
    GROUP BY job HAVING AVG(salary) > 1500;
    ```
- Find the average bonus of the employees for each department in Madrid:
  - ```
    SELECT dep_id, AVG(bonus) FROM employees
    GROUP BY dep_id HAVING dep_id = ANY (SELECT dep_id FROM departments
    WHERE location = 'Madrid');
    ```

QUERIES OVER SEVERAL TABLES (I)

- It's possible to query several tables with just one SQL sentence.
- To do that we must put the table names in the FROM clause of the main sentence or in one of the subordinate sentences.
- **Column names:**
  - Since all column names in a table are different, in an SQL sentence with just one table, writing the column name is enough to avoid ambiguities. However, when there's more than one table involved in a SQL sentence, we might have two columns with the same name. To avoid ambiguities in this situation, we must indicate to which table the offending column belongs to.
  - The table name is written immediately before the column name, and both are connected with a dot.
  - **Examples:**
    - **employees.name** refers to the employee's name.
    - **departments.name** refers to the department name.

QUERIES OVER SEVERAL TABLES (II)

- We can query several tables at the same time in two ways:
- **Put the extra tables in the FROM clause:**
  - The result is obtained combining the data of all participating tables. Tables name are written as a list and separated by commas.
  - The DBMS will perform a **cartesian product** of the tables after the FROM clause.
  - An **alias** is a local variable that can be used to give an alternative name to a column or table. These aliases are assigned inside a sentence and its usage is restricted to that particular sentence. Aliases are defined using the AS keyword, preceded by the original name and followed by the alias.
  - A table can be specified several times after a FROM clause. While doing this, using aliases is mandatory.
- **Put the extra tables in subordinated queries:**
  - These sentences are called **correlated queries**. We already saw an example of this with subordinate SELECTS used in WHERE clauses.
  - But this is not the only way of using correlated queries. A SELECT can substitute almost every value in a query. More advanced correlated queries are out of the scope of this course.

SAMPLE QUERIES OVER SEVERAL TABLES

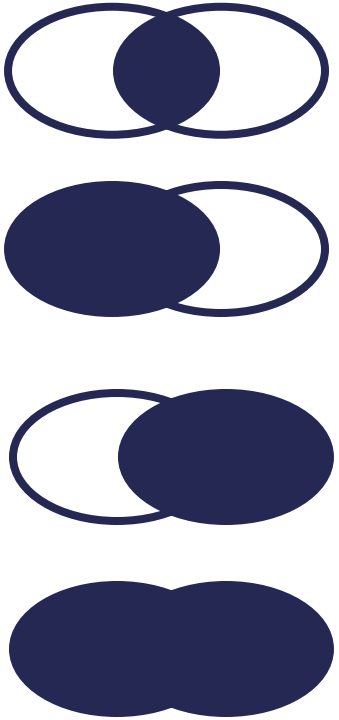
- How many employees work in Chicago?:
  - ```
    SELECT COUNT(*) FROM employees, departments
    WHERE employees.dep_id = departments.dep_id
    AND location = 'Chicago';
    ```
- Which employees work in Dallas?:
  - ```
    SELECT employees.name FROM employees, departments
    WHERE employees.dep_id = departments.dep_id
    AND location = 'Dallas';
    ```
- Show the names, job, department and location of all employees:
  - ```
    SELECT employees.name, job, departments.name, location
    FROM employees, departments
    WHERE employees.dep_id = departments.dep_id;
    ```
- Show the names of all employees and how much their managers earn:
  - ```
    SELECT emps.name, emps.manager, mans.salary
    FROM employees AS emps, employees AS mans
    WHERE mans.name = emps.manager;
    ```

JOIN OPERATIONS

- Simply putting several tables in a FROM clause is not very efficient. A cartesian product produces **a lot** of rows.
- An alternative to this approach are the **join operations**. A join operation specifies a certain rule in which tables must be combined. Joins are always written after the FROM clause:
  - ```
    FROM table1 join_type JOIN table2
    ON table1.one_column = table2.other_column
    ```
- This sentence creates a new table where all rows satisfy the condition specified after the ON clause, which compares the value of a column in the first table with the value of another column in the second table.
- Joins are **more efficient** than cartesian products. Since only the rows that match are joined.
- There are **four types** of join operations.
    ```

JOIN OPERATIONS TYPES

- **Inner Join:** `[ INNER ] JOIN`
  - All rows when there is a match in **both tables**.
- **Left Join / Left Outer Join:** `LEFT [ OUTER ] JOIN`
  - All rows from the **left table** and the matched rows from the right table.
- **Right Join / Right Outer Join:** RIGHT `[ OUTER ] JOIN`
  - All rows from the **right table** and the matched rows from the left table.
- **Outer Join:** `[ FULL ] OUTER JOIN`
  - All rows in **both tables**.



JOIN OPERATIONS EXAMPLES

- How many employees work in Chicago?:
  - ```
    SELECT COUNT(*) FROM employees INNER JOIN departments
    ON employees.dep_id = departments.dep_id
    WHERE location = 'Chicago';
    ```
- Show the names, job, department and location of all employees:
  - ```
    SELECT employees.name, job, departments.name, location
    FROM employees LEFT JOIN departments
    ON employees.dep_id = departments.dep_id;
    ```
- Show the names of all employees and how much their managers earn:
  - ```
    SELECT emps.name, emps.manager, mans.salary
    FROM employees AS mans RIGHT JOIN employees AS emps
    WHERE mans.name = emps.manager;
    ```
    ```

UNION OPERATION

- The UNION clause is used to combine the results of two or more SELECT statements:
- **Usage:**
  - `SELECT_statement1 UNION [ ALL ] SELECT_statement2;`
- By default, duplicate rows are removed. If we want to conserve the duplicate rows we will use ALL.
- To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.
- If we want to use ORDER BY, it must be put after the UNION, not in each SELECT statement.

VIEWS

- **Creating a new view:**
  - `CREATE VIEW view_name AS SELECT_clause`
- A view is a kind of “virtual table” that shows the result set of a SELECT statement. In some DBMS views are read-only.
- A view only exists when a user accesses it and always shows up-to-date data.
- Views are used to:
  - Restrict access to users.
  - Avoid duplications of data.
  - Have a convenient way of looking at common join operations.
- We can remove an existing view with the following statement:
  - `DROP VIEW view_name`

INSERTION

- **To insert new rows into a table:**
- `INSERT [ INTO ] table_name`
  `{ [ (column1 [, column2] ...) ] VALUES (value1 [, value2] ...) | SELECT_sentence }`
- A SELECT sentence can be used instead of manually inputting the values.
- **Examples:**
- Create a new department:
  - ```
    INSERT INTO departments (dep_id, name, location)
    VALUES (69, 'Marketing', 'Burgos');
    ```
- Copy into the “off sick” table all employees that are named “Charles” or “Brian”. Only the name, salary and bonus are copied.
  - ```
    INSERT INTO off_sick
    SELECT name, salary, bonus
    FROM employees
    WHERE name IN ('Charles','Brian');
    ```

UPDATE

- **To update existing rows:**
- ```
  UPDATE { table_name }
  SET column1 = { value1 | NULL | SELECT_sentene1 }
  [, column2 = { value2 | NULL | SELECT_sentence2 } ]...
  [ WHERE predicate ]
  ```
- A SELECT sentence can be used instead of manually inputting the values.
- WHERE can be used to specify which rows are changed.
- **Examples:**
- Raise the salary of all employees by 500:
  - ```
    UPDATE employees
    SET salary = salary + 500;
    ```
- Change the location of department 69 to match the location of the department 30.
  - ```
    UPDATE departments
    SET location = (SELECT location FROM departments
                    WHERE dep_id = 30)
    WHERE dep_id = 69;
    ```
    ```

DELETION

- **To remove rows from a table:**
- `DELETE [ FROM ] { table_name }`
  `[ WHERE predicate ]`
- WHERE can be used to specify which rows are deleted.
- **Examples:**
- Delete all employees:
  - `DELETE employees;`
- Delete all employees named “Peter”:
  - ```
    DELETE FROM employees
    WHERE name = 'Peter';
    ```
- Delete all employees who earn more than the average salesman salary:
  - ```
    DELETE employees
    WHERE salary > (SELECT AVG(salary) FROM employees
                    WHERE job = 'Salesman');
    ```