SQL

Rodrigo García Carmona
Universidad San Pablo-CEU
Escuela Politécnica Superior

DDL

TIPOS DE DATOS

- Todas las entradas en una columna tienen que tener el mismo tipo.
- En SQL existen muchos tipos. Destacamos los siguientes:
- **Alfanuméricos:**
  - Longitud fija: `CHAR(n)`
  - Longitud variable: `VARCHAR(n)`
  - …o más genérico: `TEXT`
- **Numéricos:**
  - `DECIMAL(precisión[,escala])`
    - Precisión: antes de la coma. Scale: después de la coma.
  - `INTEGER`
  - `REAL`
- **Fechas y horas:**
  - `TIMESTAMP:` YYYY-MM-DD HH:MI:SS
  - `DATE:` YYYY-MM-DD
  - `TIME:` HH:MI:SS
- **Otros tipos de datos:**
  - `BLOB`

TIPOS DE DATOS DE USUARIO

- **También pueden crearse tipos de datos especiales:**
- `CREATE DATATYPE custom_name data_type [ [ NOT ] NULL ] [ DEFAULT default_value ] [ CHECK ( predicate ) ]`
- Pueden tener condiciones en forma de cláusulas CHECK y valores por defecto asociados. También puede indicarse si admiten o no valores nulos. Estas condiciones son heredadas por cualquier columna definida sobre ese tipo de datos.
- Si sobre la columna se especifica cualquier otra condición, ésta prevalece sobre las condiciones del tipo de datos.
- **Ejemplos:**
  - `CREATE DATATYPE address VARCHAR(50) NULL`
    - Tipo de dato “dirección”: cadena de 50 caracteres que admite valores nulos.
  - `CREATE DATATYPE id INTEGER NOT NULL CHECK (id >= 0)`
    - Tipo de dato “id”: número entero positivo que no admite valores nulos.

CREACIÓN DE RELACIONES

- **Antes de añadir entradas es preciso crear una relación:**
- ```
  CREATE TABLE table_name (
  column_definition [ [ NOT ] NULL ] [ DEFAULT default_value ]
  [ [ CONSTRAINT name ] column_constraint ],
  ...
  [ [ CONSTRAINT name ] table_constraint ],
  ...)
  ```
- **Definición de columna:**
  - `column_name data_type`
  - Define una columna de la tabla. Los tipos de datos permitidos son los descritos anteriormente. Dos columnas en la misma tabla no pueden tener el mismo nombre.
  - Si se especifica NOT NULL, o si se le aplica una restricción PRIMARY KEY, la columna no puede tomar valores nulos.
  - **Valores por defecto posibles:**
    - `character_string, number`
    - `CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP`
    - `NULL`
  ```

RESTRICCIONES DE TABLA

- **Sobre una tabla pueden aplicarse las siguientes restricciones:**
- `UNIQUE ( column_name, ... )`
  - Identifica una o más columnas que identifican unívocamente cada fila de la tabla.
- `PRIMARY KEY ( nombre_columna, ... )`
  - Igual que UNIQUE + NOT NULL, pero una tabla sólo puede tener una clave primaria.
- `CHECK ( predicate )`
  - Impone condiciones que deben cumplir los valores de una columna. Por ejemplo, podría utilizarse una restricción de este tipo para asegurarse de que la columna "sexo" sólo toma los valores "hombre" o "mujer".
- `FOREIGN KEY [ alias ] [ ( column_name, ... ) ]`
  `REFERENCES table_name [ ( column_name, ... ) ]`
  `[ actions ]`
  - Restringe los valores para un conjunto de columnas para que deban coincidir con los valores de la clave principal o una columna UNIQUE de otra tabla. A esta restricción se le llama **clave ajena** (FOREIGN KEY).
  - Normalmente, una clave ajena hace referencia a una clave primaria.

RESTRICCIONES DE COLUMNA

- Muy similares a las restricciones de tabla. De hecho, es otra forma de expresar la misma información.
- Si la restricción sólo afecta a una columna es mejor usar restricciones de columna.
- **Sobre una columna pueden aplicarse las siguientes restricciones:**
  - `UNIQUE`
  - `PRIMARY KEY`
    - En SQLite, una PRIMARY KEY debería ser INTEGER.
  - `REFERENCES table_name [ ( column_name ) ] [ actions ]`
    - Con restricciones de columna no tenemos que escribir FOREIGN KEY.
  - `CHECK ( predicate )`
- Las siguientes sentencias son equivalentes:
  - `CREATE TABLE town (code INTEGER UNIQUE);`
  - `CREATE TABLE town (code INTEGER, UNIQUE (code));`

AUTOINCREMENT

- **AUTOINCREMENT** es una restricción especial, que sólo puede ser usada en ciertas soluciones SQL.
- SQLite soporta AUTOINCREMENT sólo en las columnas que:
  - Son del tipo INTEGER.
  - Tienen la restricción PRIMARY KEY.
- En un INSERT, si no se le da un valor explícitamente a la columna PRIMARY KEY, ésta tomará automáticamente un valor que no esté siendo usado, normalmente el siguiente al mayor que esté siendo usado en ese momento.
- AUTOINCREMENT impide que se reusen valores durante **toda la vida** de la base de datos. O, dicho de otra forma, evitar la reutilización de valores que tuvieran filas que hayan sido borradas.

ACCIONES

- Podemos definir las acciones a tomar para mantener las relaciones de claves ajenas. Cuando se modifica o se borra una clave principal en una tabla, los valores correspondientes de claves ajenas en otras tablas deben modificarse igualmente.
- En SQLite es necesario activarlas escribiendo la sentencia: **PRAGMA foreign_keys = ON**
- **Acciones:**
  - `[ ON UPDATE action_type ] [ ON DELETE action_type ]`
- **Tipos de acciones:**
  - CASCADE
    - Usado con ON UPDATE, modifica las claves ajenas al nuevo valor de la clave principal.
    - Usado con ON DELETE, borra las filas cuyo valor de clave ajena coincide con el valor de la clave principal borrada.
  - SET NULL
    - Pone a NULL todos los valores de clave ajena que se correspondan con el valor de la clave principal borrada o modificada.
  - SET DEFAULT
    - Asigna el valor especificado por defecto por la cláusula DEFAULT para todos los valores de clave ajena que se correspondan con el valor de la clave principal borrada o modificada.
  - RESTRICT
    - Impide la modificación o el borrado de la clave principal si existen valores de claves ajenas que la referencian.

EJEMPLOS DE CREACIÓN DE TABLAS

- Crea una tabla con las columnas dep (clave primaria), name y location, permitiendo que name y location tomen valores nulos:
  - ```
    CREATE TABLE departments (
    dep_id INTEGER PRIMARY KEY,
    name TEXT,
    location TEXT
    );
    ```
- Crea una tabla de empleados que hace referencia a la clave primaria de departamentos como clave ajena, especificando las acciones SET NULL en borrado y CASCADE en actualización:
  - ```
    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);
    ```

ÍNDICES

- **Creando un índice:**
  - ```
    CREATE [ UNIQUE ] INDEX index_name
    ON table_name ( column_name [ ASC | DESC ], ... )
    ```
- Los índices son tablas especiales que el gestor utilizar para acelerar el acceso a datos. Son como los índices de un libro, y el gestor los usa automáticamente. Aceleran los SELECTs, pero enlentecen los UPDATEs e INSERTs.
- Una vez que se ha creado un índice, no vuelve a referenciarse salvo para borrarlo.
- La restricción UNIQUE asegura que no habrá dos filas en la tabla con valores idénticos para las columnas del índice.
- Por defecto se crea en orden ascendente, pero pueden crearse descendentes usando DESC.
- El gestor crea automáticamente índices para las claves primarias y para las columnas con restricciones UNIQUE.
- **Ejemplo:**
  - ```
    CREATE UNIQUE INDEX ix_emp ON employees (manager);
    ```

MODIFICACIÓN DE LA ESTRUCTURA DE UNA TABLA

- Podemos modificar una tabla ya creada.
- **Modificando una tabla:**
- ```
  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 }
  ```
- **Ejemplos:**
  - Añadir una nueva columna con el apellido a la tabla de empleados.
    - ```
      ALTER TABLE employees
      ADD surname TEXT;
      ```
  - Añadir una nueva restricción a la tabla de empleados. El manager debe ser otro empleado:
    - ```
      ALTER TABLE employees
      ADD CONSTRAINT mgr_const FOREIGN KEY (manager) REFERENCES
      employees(name);
      ```
- SQLite **sólo soporta** las variantes RENAME TO y ADD COLUMN de ALTER TABLE.
      ```

BORRADO DE OBJETOS

- Permite borrar un objeto ya creado.
- **Uso:**
  - `DROP [ DATATYPE | INDEX | CONSTRAINT | TABLE ] object_name`
- **Ejemplos:**
  - Borrado de tipo de datos:
    - `DROP DATATYPE address;`
  - Borrado de índice:
    - `DROP INDEX employees.ix_emp;`
  - Borrado de restricción (¡debe tener nombre!):
    - `DROP CONSTRAINT employees.const_emp;`
  - Borrado de tabla:
    - `DROP TABLE employees;`

DML

CONSULTAS

- **Búsqueda de resultados en una base de datos:**
- ```
  SELECT [ DISTINCT ] { expr_1 [ , expr_2 ] ... | * }
  FROM table_name
  [ WHERE predicate ]
  [ ORDER BY column1 [ , column2 ]... [ ASC | DESC ]
  ```
- Tras la cláusula SELECT se escriben expresiones (habitualmente nombres de columnas) pertenecientes a la tabla cuyo nombre aparece en la cláusula FROM. Si en lugar de nombres de columnas se pone un * equivale a escribir los nombres de todas las columnas de la tabla.
- El resultado de la ejecución de una sentencia SELECT es siempre otra tabla. Las columnas de la tabla resultante serán las que figuren tras la cláusula SELECT, en el mismo orden.
- Si se usa DISTINCT se eliminarán los resultados de búsqueda repetidos.
- Sólo se devolverán los resultados que cumplan con la condición indicada en WHERE.
- Con ORDER BY pueden ordenarse los resultados, ascendiente (por defecto) o descendientemente. Si no se especifica se devuelven en cualquier orden. También se puede usar el número de la columna según ha sido especificada.
  ```

EJEMPLOS DE CONSULTAS

- Lista todos los empleados con sus datos asociados.
  - `SELECT * FROM employees;`
- Lista todos los empleados, pero mostrando sólo sus nombres y managers:
  - `SELECT name, manager FROM employees;`
- Lista los nombres de todos los empleados, ordenados por el nombre de su manager:
  - `SELECT name FROM employees ORDER BY manager;`
- Lista el nombre, manager y departamento de todos los empleados, ordenando descendientemente en primer lugar por manager y después por departamento:
  - `SELECT name, manager, dep_id FROM employees ORDER BY 2, 3 DESC;`
- Mostrar todos los managers:
  - `SELECT DISTINCT manager FROM employees;`

EXPRESIONES

- Las expresiones sirven para realizar operaciones sobre los datos en una consulta. Por ejemplo, se puede solicitar el resultado del producto de los valores de dos columnas, o el valor de una columna dividido por un valor.
- También pueden utilizarse expresiones en los predicados que aparecen en la cláusula WHERE.
- Una expresión es una combinación de **operadores**, **operandos** y **paréntesis**. El resultado de la ejecución de una expresión es un único valor.
- Los **operandos** pueden ser nombres de columnas, constantes u otras expresiones.
- **Operadores sobre datos numéricos:**
  - Suma: +
  - Resta: -
  - Multiplicación: *
  - División: /
- Los operadores sobre datos alfanuméricos dependen de la implementación de SQL.

EJEMPLOS DE EXPRESIONES

- Una expresión puede estar seguida de una cadena de caracteres, que aparecerá como el nombre de la columna en el resultado de la consulta.
- **Ejemplos:**
  - Calcular el sueldo anual a percibir por cada empleado y darle el nombre "Annual Salary":
    - `SELECT name, salary * 4 * 12 'Annual Salary' FROM employees;`
  - Mostrar el nombre del empleado y una columna que contenga el sueldo junto con las bonificaciones. El nuevo sueldo se mostrará como "Salary with Bonus" :
    - `SELECT name, salary + bonus 'Salary with Bonus' FROM employees;`

PREDICADOS

- Hasta el momento hemos visto consultas básicas, pero podemos especificar condiciones de búsqueda más elaboradas.
- Un **predicado** es una expresión lógica sobre valores de columnas, y puede devolver “true", "false" o “unknown"
- Para expresar predicados usamos:
  - Claúsulas CHECK en el DDL.
  - Claúsulas WHERE en el DML.
- Sólo se considera satisfecha la condición de búsqueda expresada en un predicado cuando toma el valor “true". Se rechazarán las filas para las que tome el valor “false" o “unknown".
- **Predicados básicos:** Comparación entre dos valores.
  - x = y : “true” si x es igual a y.
  - x <> y : “true” si x no es igual a y.
  - x < y : “true” si x es menor que y.
  - x > y : “true” si x es mayor que y.
  - x <= y : “true” si x es menor o igual que y.
  - x >= y : “true” si x es mayor o igual que y.

EJEMPLOS DE PREDICADOS

- Muestra todos los empleados que trabajan como vendedores:
  - ```
    SELECT name
    FROM employees
    WHERE job = 'salesman';
    ```
- Muestra los empleados que no trabajen en el departamento 30:
  - ```
    SELECT name
    FROM employees
    WHERE dep_id <> 30;
    ```
- Muestra los empleados que fueron contratados antes del 1 de Enero de 1982:
  - ```
    SELECT name
    FROM employee
    WHERE hiredate < '1982-1-1';
    ```
- Muestra el nombre y la fecha de alta en la empresa de todos los analistas:
  - ```
    SELECT name, hiredate
    FROM employees
    WHERE job = 'analyst';
    ```

SENTENCIAS SUBORDINADAS

- El segundo operando de un predicado puede ser, en lugar de una expresión, el resultado de la ejecución de otra sentencia SELECT, a la que se llama **sentencia subordinada (subordinate query)**.
- Deberá ir entre paréntesis y devolver como resultado **un único** valor. Es decir, la tabla resultante debe tener una sola columna y una fila o ninguna.
- Si el resultado de esta sentencia SELECT es una tabla vacía, su valor se considera “unknown".
- No se puede usar la cláusula ORDER BY en una sentencia subordinada.

EJEMPLOS DE SENTENCIAS SUBORDINADAS

- Mostrar los empleados cuyo salario sea superior al de Adams:
  - ```
    SELECT name FROM employees
    WHERE salary > (SELECT salary FROM employees
                    WHERE name = 'Adams');
    ```
- Mostrar los empleados que trabajan en el mismo departamento que Clark:
  - ```
    SELECT name FROM employees
    WHERE dep_id = (SELECT dep_id FROM employees
                    WHERE name = 'Clark');
    ```
- Mostrar los empleados cuyo jefe es Blake:
  - ```
    SELECT name FROM employees
    WHERE manager = (SELECT name FROM employees
                     WHERE name = 'Blake');
    ```
    ```

PREDICADOS COMPUESTOS

- Los **predicados compuestos** son combinaciones de predicados usando los operadores lógicos AND, OR y NOT.
- AND y OR se usan con dos predicados, mientras que NOT se usa sólo con uno.
- Pueden utilizarse paréntesis.
- **Ejemplos:**
- Mostrar los nombres de los vendedores que ganen más de 1500:
  - ```
    SELECT name FROM employees
    WHERE job = 'salesman' AND salary > 1500;
    ```
- Mostrar el nombre de los empleados que trabajen como dependientes o en el departamento 30:
  - ```
    SELECT name FROM employees
    WHERE job = 'clerk' OR dep_id = 30;
    ```

PREDICADOS Y CUANTIFICADORES

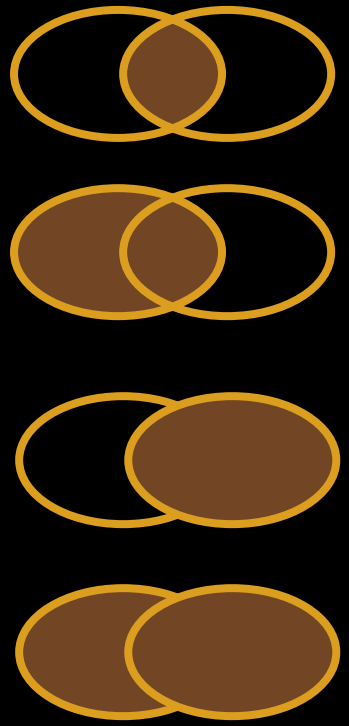
- **Cuantificador universal (para todo):**
  - **ALL:** Es verdadero si la condición es verdadera para todos y cada uno de los valores devueltos por la SELECT subordinada.
  - **Ejemplo:** Mostrar el nombre de los empleados que ganen más que todos los vendedores:
    - ```
      SELECT name FROM employees
      WHERE salary > ALL (SELECT salary FROM employees
                          WHERE job = 'salesman');
      ```
- **Cuantificador existencial (existe):**
  - **SOME / ANY:** Es verdadero si la condición es verdadera para uno cualquiera de los valores devueltos por la ejecución de la sentencia SELECT subordinada.
  - SOME y ANY son equivalentes.
  - **Ejemplo:** Mostrar el nombre de los empleados que ganen más que alguno de los vendedores:
    - ```
      SELECT name FROM employees
      WHERE salary > SOME (SELECT salary FROM employees
                           WHERE job = 'salesman');
      ```

COMPARACIONES EN PREDICADOS

- **Comprobación de valor nulo:**
  - `column_name IS [ NOT ] NULL`
  - Elimina los registros con un valor null (o distinto a null) de los resultados.
  - **Ejemplo:** Mostrar los empleados que tienen bonificación:
    - `SELECT name FROM employees WHERE bonus IS NOT NULL;`
- **Comprobación de pertenencia a un conjunto:**
  - `predicate [ NOT ] IN ( value_1 [ , value_2 ] ... )`
  - Averigua si el resultado de la evaluación de una expresión está incluido en la lista de valores especificada tras la palabra IN.
  - **Ejemplo:** Mostrar los empleados llamados “Julius”, “Allen” o “Scott”:
    - ```
      SELECT * FROM employees
      WHERE name IN (‘Julius’, ‘Allen’, ‘Scott’);
      ```
      ```

COMPARAR PREDICADOS CON VALORES

- **BETWEEN - AND:**
  - `predicate_1 [ NOT ] BETWEEN predicate_2 AND predicate_3`
  - Para comprobar si un valor está comprendido entre otros dos (ambos inclusive), o no.
  - **Ejemplo:** Mostrar los empleados cuyo sueldo esté entre 2000 y 3000:
    - `SELECT name FROM employees WHERE salary BETWEEN 2000 AND 3000;`
- **LIKE:**
  - `column_name [ NOT ] LIKE pattern`
  - Para buscar combinaciones de caracteres que coincidan con un patrón especificado.
  - La constante alfanumérica puede contener cualquier carácter válido.
  - En el patrón, "_" se sustituye por cualquier carácter, pero siempre uno; "%" se sustituye por una cadena cualquiera de cualquier longitud (incluida la cadena vacía); “[lista]” se sustituye por cualquier carácter de la lista; “[!lista]” se sustituye por cualquier carácter que no esté en la lista.
  - **Ejemplo:** Mostrar los empleados cuyo nombre tenga como segunda letra una “d”:
    - `SELECT name FROM employees WHERE name LIKE ‘_d%’;`

FUNCIONES DE COLUMNAS

- Devuelven un solo valor tras aplicar una determinada operación a los valores contenidos en una o más columnas.
- Su argumento es una colección de valores tomados de una o más columnas.
- Funciones de columnas:
  - **AVG** devuelve la media de los valores de la colección.
  - **MAX** devuelve el valor máximo de la colección.
  - **MIN** devuelve el valor mínimo de la colección.
  - **SUM** devuelve la suma de todos los valores en la colección.
  - **COUNT** devuelve el número de elementos en la colección.
- El argumento suele ser una expresión. Las funciones de columnas tambien pueden utilizarse como operandos en los predicados de las claúsulas WHERE.
- Como pasa previo a ejecutar estas funciones, existe la posibilidad de dividir las filas en varios grupos usando la claúsula GROUP BY. Las funciones de columna se ejecutan sobre cada uno de los grupos por separado.
- Antes de ejecutar esta funciones se pueden eliminar registros no deseados usando la claúsula HAVING.

EJEMPLOS DE FUNCIONES DE COLUMNAS

- Calcular el número de empleados que tienen bonificación y la bonificación media:
  - `SELECT COUNT(bonus), AVG(bonus) FROM employees;`
- Encontrar el sueldo medio de los analistas:
  - `SELECT AVG(salary) 'Average Salary' FROM employees WHERE job = 'analyst';`
- Encontrar el sueldo más alto, el más bajo y la diferencia entre ambos:
  - `SELECT MAX(salary), MIN(salary), MAX(salary)-MIN(salary) FROM employees WHERE job = 'analyst';`
- Hallar el numero de trabajos distintos que existen en el departamento 30:
  - `SELECT COUNT(DISTINCT job) FROM employees WHERE dep_id = 30;`
- Calcular cuánto invierte anualmente en sueldos la compañía:
  - `SELECT SUM(salary * 12) 'Expenses' FROM employees;`
- Encontrar el nº de trabajadores en el departamento 30 que ganan más que el vendedor medio:
  - `SELECT COUNT(name) FROM employees WHERE dep_id = 30 AND salary > (SELECT AVG(salary) FROM employees WHERE job = 'salesman');`

AGRUPAMIENTO DE FILAS PARA FUNCIONES DE COLUMNAS

- Podemos agrupar filas antes de ejecutar funciones de columnas. Al hacerlo la función se ejecutará sobre cada grupo por separado.
- **GROUP BY:** Cláusula opcional de la sentencia SELECT que sirve para agrupar filas. Debe aparecer después de la cláusula WHERE, si ésta existe.
- **Uso:**
  - `GROUP BY column_name_1 [ , column_name_2] ...`
- **Ejemplos:**
- Mostrar los sueldos mínimo y máximo de los empleados, agrupados por ttrabajo:
  - ```
    SELECT job, MIN(salary), MAX(salary) FROM employees
    GROUP BY job;
    ```
- Calcular el número de empleados de cada departamento que tienen un ssueldo superior a la media:
  - ```
    SELECT dep_id, COUNT(name) FROM employees
    WHERE salary > (SELECT AVG(salary) FROM employees)
    GROUP BY dep_id;
    ```

DESCARTE DE FILAS EN FUNCIONES DE COLUMNAS

- También podemos descartar filas no deseadas antes de ejecutar una función de columnas. WHERE no puede utilizarse con GROUP BY.
- **HAVING:** Cláusula opcional de la sentencia SELECT utilizada para descartar filas que no cumplen con unas condiciones especificadas. Se usa con GROUP BY.
- **Utilización:**
  - `HAVING expression`
- **Ejemplos:**
- Encontrar el sueldo medio y mínimo de cada puesto, considerando sólo aquellos puestos cuyo sueldo medio esté por encima de 1500:
  - ```
    SELECT job, AVG(salary), MIN(salary) FROM employees
    GROUP BY job HAVING AVG(SAL) > 1500;
    ```
- Mostrar la media de la bonificación por departamento, considerando sólo aquellos empleados con un sueldo de al menos 1000:
  - ```
    SELECT AVG(bonus) FROM employees
    GROUP BY dep_id HAVING salary > 1000;
    ```

CONSULTAS SOBRE VARIAS TABLAS (I)

- Es posible hacer una consulta sobre varias tablas con una sola sentencia SQL.
- Para ello ponemos los nombres de dichas tablas en la cláusula FROM de la sentencia principal o de alguna de sus subordinadas.
- **Nombres de columnas:**
  - Como los nombres de las columnas no pueden repetirse dentro de una tabla, en una sentencia SQL con sólo una tabla basta con poner el nombre de la columna para evitar ambigüedades. Sin embargo, cuando hay más de una tabla en una sentencia SQL cabe la posibilidad de que haya dos columnas con el mismo nombre. En estos casos, para referirnos a una columna determinada, hay que indicar a qué tabla pertenece ésta.
  - Para ello escribimos el nombre de la columna precedido del de la tabla y separados por un punto.
  - **Ejemplos:**
    - **employees.name:** el nombre de los empleados.
    - **departments.name:** el nombre de los departamentos.

CONSULTAS SOBRE VARIAS TABLAS (II)

- Podemos hacer una consulta sobre varias tablas de dos formas:
- **Poner los nombres de todas las tablas en la claúsula FROM:**
  - El resultado se obtiene combinando los datos de todas las tablas intervinientes. Los nombres de la tabla se escriben en una lista, separados por comas.
  - El DBMS llevará a cabo un **producto cartesiano** de las tablas especificadas en el FROM.
  - Un **alias** es una variable local que podemos utilizar para dar un nombre alternativo a una columna o una tabla. Estos alias son asignados dentro de una sentencia y están limitados a ella. Para definir un alias usamos la palabra clave AS, precedida del nombre original de la tabla y seguida del alias.
  - Una tabla puede aparecer más de una vez en la misma cláusula FROM. En este caso es necesario asignar alias.
- **Poner los nombres de las tablas adicionales en sentencias subordinadas:**
  - A este tipo de consultas las llamamos **consultas correlacionadas**. Ya hemos visto un ejemplo de esto con SELECTs subordinados dentro de claúsulas WHERE.
  - Pero ésta no es la única forma de usar consultas correlacionadas. Un SELECT puede sustituir prácticamente cualquier valor dentro de una consulta. Existen consultas correlacionadas más avanzadas, pero no las veremos en esta asignatura.

EJEMPLOS DE CONSULTAS SOBRE VARIAS TABLAS

- ¿Cuántos empleados trabajan en ‘CHICAGO’ ?:
  - ```
    SELECT COUNT(*) FROM employees, departments
    WHERE employees.dep_id = departments.dep_id
    AND location = ‘Chicago’;
    ```
- ¿Qué empleados trabajan en ‘DALLAS’?:
  - ```
    SELECT employees.name FROM employees, departments
    WHERE employees.dep_id = departments.dep_id
    AND location = ‘Dallas’;
    ```
- Mostrar nombres, empleo, departamento y lugar de trabajo de todos los empleados:
  - ```
    SELECT employees.name, job, departments.name, location
    FROM employees, departments
    WHERE employees.dep_id = departments.dep_id;
    ```
- Mostrar para cada empleado su nombre, el de su jefe, y cuánto gana su jefe:
  - ```
    SELECT emps.name, emps.manager, mans.salary
    FROM employees AS emps, employees AS mans
    WHERE mans.name = emps.manager;
    ```

OPERACIÓN DE REUNIÓN (JOIN)

- Limitarse a poner varias tablas en una claúsula FROM no es muy eficiente. Un producto cartesiano genera **un montón** de filas.
- Una alternativa es utilizar la **operación de reunión (join)**. En un join especificamos la regla que el DBMS seguirá para combinar las tablas. Los joins siempre se escriben tras la claúsula FROM.
  - ```
    FROM table1 join_type JOIN table2
    ON table1.one_column = table2.other_column
    ```
- Esta sentencia crea una nueva tabla en la que todas las filas cumplen con la condición especificada en la claúsula ON, que compara el valor de una columna en la primera tabla con el valor de otra en la segunda tabla.
- Los joins son **más eficientes** que los productos cartesianos, ya que sólo las filas que cumplen con la condición son emparejadas.
- Hay **cuatro tipos** de joins.
    ```

TIPOS DE JOINS

- **Inner Join:** `[ INNER ] JOIN`
  - Todas las filas en las que haya un emparejamiento de resultados en **ambas tablas**.
- **Left Join / Left Outer Join:** `LEFT [ OUTER ] JOIN`
  - Todas las filas de la **tabla izquierda** y las emparejadas de la tabla derecha.
- **Right Join / Right Outer Join:** RIGHT `[ OUTER ] JOIN`
  - Todas las filas de la **tabla derecha** y las emparejadas de la tabla izquierda.
- **Outer Join:** `[ FULL ] OUTER JOIN`
  - Todas las filas en **ambas tablas**.



EJEMPLOS DE JOINS

- ¿Cuántos empleados trabajan en Chicago?:
  - ```
    SELECT COUNT(*) FROM employees INNER JOIN departments
    ON employees.dep_id = departments.dep_id
    WHERE location = 'Chicago';
    ```
- Mostrar el nombre, puesto, departamento y lugar de trabajo de todos los empleados:
  - ```
    SELECT employees.name, job, departments.name, location
    FROM employees LEFT JOIN departments
    ON employees.dep_id = departments.dep_id;
    ```
- Mostrar los nombres de todos los empleados y cuánto ganan sus jefes:
  - ```
    SELECT emps.name, emps.manager, mans.salary
    FROM employees AS mans RIGHT JOIN employees AS emps
    WHERE mans.name = emps.manager;
    ```
    ```

OPERACIÓN DE UNIÓN

- La claúsula UNION sirve para combinar los resultados de dos o más sentencias SELECT.
- **Utilización:**
  - `SELECT_statement1 UNION [ ALL ] SELECT_statement2;`
- Por defecto se eliminan las filas duplicadas. Pero si queremos conservarlas podemos añadir ALL.
- Para usar UNION todos los SELECT deben de tener el mismo número de columnas, el mismo número de expresiones de columna, los mismos tipos de dato y estar en el mismo orden. No tienen por qué tener la misma longitud.
- Si queremos usar ORDER BY debemos ponerlo tras el UNION completo, no en cada sentencia SELECT.

VISTAS

- **Crear una nueva vista:**
  - `CREATE VIEW view_name AS SELECT_clause`
- Una vista es una especie de “tabla virtual”, que muestra los resultados de la ejecución de una sentencia SELECT. En algunos DBMS las vistas son de sólo lectura.
- Una vista sólo existe cuando un usuario accede a ella y siempre muestra información actualizada.
- Las vistas se usan para:
  - Restringir el acceso a ciertos usuarios.
  - Evitar la duplicación de información.
  - Tener una forma cómoda de acceder a joins que hagamos con frecuencia.
- Podemos eliminar una vista creada anteriormente escribiendo:
  - `DROP VIEW view_name`

INSERCIÓN

- **Inserción de filas en tablas:**
- `INSERT [ INTO ] table_name`
  `{ [ (column1 [, column2] ...) ] VALUES (value1 [, value2] ...) | SELECT_sentence }`
- Puede usarse una sentencia SELECT en lugar de introducir los valores a mano.
- **Ejemplos:**
- Crear un nuevo departamento:
  - `INSERT INTO departments (dep_id, name, location)`
    `VALUES (69, 'Marketing', 'Burgos');`
- Copiar a la tabla "off sick" (de baja) todos los empleados que se llamen "Charles" o "Brian". Sólo el nombre, el sueldo y la bonificación se copian.
  - `INSERT INTO off_sick`
    `SELECT name, salary, bonus`
    `FROM employees`
    `WHERE name IN ('Charles','Brian');`

MODIFICACIÓN

- **Modificación de valores en tablas:**
- ```
  UPDATE { table_name }
  SET column1 = { value1 | NULL | SELECT_sentene1 }
  [, column2 = { value2 | NULL | SELECT_sentence2 } ]...
  [ WHERE predicate ]
  ```
- Puede usarse una sentencia SELECT en lugar de introducir los valores a mano.
- Puede usarse WHERE para especificar qué filas son modificadas.
- **Ejemplos:**
- Aumentar el sueldo de todos los empleados en 500:
  - ```
    UPDATE employees
    SET salary = salary + 500;
    ```
- Cambiar la ubicación del departamento 69 para que coincida con la del 30.
  - ```
    UPDATE departments
    SET location = (SELECT location FROM departments
                    WHERE dep_id = 30)
    WHERE dep_id = 69;
    ```
    ```

BORRADO

- **Borrado de filas en tablas:**
- `DELETE [ FROM ] { table_name }`
  `[ WHERE predicate ]`
- Puede usarse un WHERE para especificar qué filas son borradas.
- **Ejemplos:**
- Borrar todos los empleados:
  - `DELETE employees;`
- Borras todos los empleados que se llamen “Peter”:
  - ```
    DELETE FROM employees
    WHERE name = ‘Peter';
    ```
- Borrar todos los empleados que ganen más que el vendedor medio:
  - ```
    DELETE employees
    WHERE salary > (SELECT AVG(salary) FROM employees
                    WHERE job = ‘Salesman');
    ```