

Principles of Databases

SQL

David Sinclair

SQL

- SQL stands for *Standard Query Language*.
- SQL is an ANSI standard language for interacting with relational database management systems.
- SQL is a declarative language and its statements belong to one of two categories:
 - **DDL** The *Data Definition Language* describes the structure of the database, i.e. what tables make up the database and what attributes make up each table.
 - **DML** The *Data Manipulation Language* defines the operations that can be performed on the data within the database.
- SQL is compositional.
 - SQL statements operate on relations.
 - The result of an SQL operation is another relation, called the *result-set*.

Tables

- The relations in a relational database are stored as tables.
- Each row in a table stores an instance of the relation.
- The columns of the table represent the attributes of the relation.
- Each column has a name and a datatype. Some typical datatypes are:

CHAR()	A fixed string from 0 to 255 characters long.
VARCHAR()	A variable string from 0 to 255 characters long.
TEXT	A string with a maximum length of 65535 characters.
INT()	-2147483648 to 2147483647
FLOAT	A small number with a floating decimal point.
DOUBLE(,)	A large number with a floating decimal point.

SELECT Statement

- The syntax of the SELECT statement is
 SELECT $attr_1, attr_2, \dots, attr_n$
 FROM $table_1, table_2, \dots, table_m$
 WHERE $condition$;
 1. In the FROM clause $table_1, table_2, \dots, table_m$ is the cross product of the tables. $T_1 \times T_2$ generates a table where each row of T_1 is combined with each row of T_2 .
 2. The WHERE clause keeps the rows in which $condition$ evaluates to *true*.
 3. In the SELECT clause, $attr_1, attr_2, \dots, attr_n$ specifies the columns to keep.
- Consider the tables

Person		
ID	Name	countryID
1	John	1
2	Gina	3

Country		
countryID	Name	Capital
1	Ireland	Dublin
2	China	Beijing
3	Italy	Rome

SELECT Statement (2)

SELECT * FROM Person, Country;

yields:

ID	Name	countryID	countryID	Name	Capital
1	John	1	1	Ireland	Dublin
2	Gina	3	1	Ireland	Dublin
1	John	1	2	China	Beijing
2	Gina	3	2	China	Beijing
1	John	1	3	Italy	Rome
2	Gina	3	3	Italy	Rome

Note:

- * selects all columns.
- Column names can get duplicated.
- SQL keywords are not case sensitive.

SELECT * FROM Person, Country

WHERE Person.countryID = Country.countryID;

yields:

ID	Name	countryID	countryID	Name	Capital
1	John	1	1	Ireland	Dublin
2	Gina	3	3	Italy	Rome

SELECT Statement (3)

SELECT Person.Name, Country.Name, Capital

FROM Person, Country

WHERE Person.countryID = Country.countryID;

yields:

Name	Name	Capital
John	Ireland	Dublin
Gina	Italy	Rome

SELECT Person.Name, Country.Name AS Nation, Capital

FROM Person, Country

WHERE Person.countryID = Country.countryID;

yields:

Name	Nation	Capital
John	Ireland	Dublin
Gina	Italy	Rome

WHERE clause

- The WHERE clause is used to extract records that satisfy a specified condition. This condition can use the following operators:
 - = (equals), < > (not equal to)
 - > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to)
 - BETWEEN (Between an inclusive range)
 - LIKE (Search for a pattern)
 - IN (Allows multiple distinct values in a WHERE clause)
- SQL uses single quotes to enclose text values (though most database systems also accept double quotes).

```
SELECT * FROM Country WHERE Name = 'Italy';
```

WHERE clause - AND & OR Operators

Let us change the example tables:

Person

ID	Name	Age	countryID
1	John	20	1
2	Gina	25	3
3	Ping	22	2
4	Wei	21	2
5	Ann	19	1

Country

countryID	Name	Capital
1	Ireland	Dublin
2	China	Beijing
3	Italy	Rome
4	France	Paris

- The AND operator is true if both its sub-conditions are true.

```
SELECT *
FROM Person, Country
WHERE Person.countryID = Country.countryID AND Country.Name =
'China';
```

ID	Name	Age	countryID	countryID	Name	Capital
3	Ping	22	2	2	China	Beijing
4	Wei	21	2	2	China	Beijing

WHERE clause - AND & OR Operators (2)

- The OR operator is true if one or both of its sub-conditions are true.

```
SELECT *
FROM Person
WHERE Name = 'John' OR Name = 'Gina';
```

ID	Name	Age	countryID
1	John	20	1
2	Gina	25	3

- The AND & OR operators can be combined.

```
SELECT ID, Person.Name, Country.Name, Capital
FROM Person, Country
WHERE Person.countryID = Country.countryID AND
(Person.Name = 'Wei' OR Person.Name = 'Ann');
```

ID	Name	Name	Capital
4	Wei	China	Beijing
5	Ann	Ireland	Dublin

ORDER BY

- The ORDER BY keyword is used to sort the *result-set* by a specified column.
- The columns can be sorted in ascending (ASC) or descending (DESC) order. Ascending order is the default ordering.

```
SELECT *
FROM Person
ORDER BY Name;
```

ID	Name	Age	countryID
5	Ann	19	1
2	Gina	25	3
1	John	20	1
3	Ping	22	2
4	Wei	21	2

```
SELECT *
FROM Person
ORDER BY Age DESC;
```

ID	Name	Age	countryID
2	Gina	25	3
3	Ping	22	2
4	Wei	21	2
1	John	20	1
5	Ann	19	1

LIKE

- The SQL syntax for the LIKE operator is:
 SELECT *column_names*
 FROM *table_names*
 WHERE *column_name* LIKE *pattern*
- The *pattern* in the LIKE operator can use the following wildcards:

% zero or more characters
 _ exactly one character
 [charlist] Any single character in charlist
 [!charlist] Any single character **not** in charlist

```
SELECT *
FROM Country
WHERE Name LIKE [!i]%;
```

countryID	Name	Capital
2	China	Beijing
4	France	Paris

IN

- The IN operator lets you specify multiple values in a WHERE clause.
- The SQL syntax for the IN operator is:

```
SELECT column_names
FROM table_names
WHERE column_name IN (value1,value2,...);
```

```
SELECT ID, Person.Name, Country.Name, Capital
FROM Person, Country
WHERE Person.countryID = Country.countryID AND
Country.Name IN ('China', 'Italy');
```

ID	Name	Name	Capital
2	Gina	Italy	Rome
3	Ping	China	Beijing
4	Wei	China	Beijing

```
SELECT ID, Person.Name, Country.Name, Capital
FROM Person, Country
WHERE Person.countryID = Country.countryID AND
Country.Name NOT IN ('China', 'Ireland', 'France');
```

ID	Name	Name	Capital
2	Gina	Italy	Rome

IN (2)

- The IN operator can be used in nested SELECT statements.

```
SELECT Model
FROM Product
WHERE ManufacturerID IN
    (SELECT ManufacturerID
     FROM Manufacturer
     WHERE Manufacturer.Name IN ('Apple','IBM'))
);
```

BETWEEN

- The BETWEEN operator is used to select from a range of values.
- The SQL syntax for the BETWEEN operator is:

```
SELECT column_names
FROM table_names
WHERE column_name BETWEEN value1 AND value2;

SELECT *
FROM Person
WHERE Age Between 20 AND 23;
```

ID	Name	Age	countryID
1	John	20	1
3	Ping	22	2
4	Wei	21	2

- Warning:** Though they should not, different DBMS handle the BETWEEN operator differently. Be careful!
 - Some include *value*₁ and *value*₂ in the test range. This is the standard definition of BETWEEN.
 - Some exclude *value*₁ and *value*₂ from the test range.
 - Some include *value*₁ but exclude *value*₂ from the test range.

SELECT TOP

- The TOP clause can be very useful especially when a query on a large database generates a result-set with hundreds or thousands of tuples. Large result-sets can impact on the performance of nested queries.

- The SQL syntax for the TOP clause is:

```
SELECT TOP number{number PERCENT} column_name(s)
FROM table_name(s);
```

```
SELECT TOP 3 *
FROM Person
ORDER BY Age;
```

ID	Name	Age	countryID
5	Ann	19	1
1	John	20	1
4	Wei	21	2

```
SELECT TOP 40 PERCENT *
FROM Person
ORDER BY Age DESC;
```

ID	Name	Age	countryID
2	Gina	25	3
3	Ping	22	2

SELECT DISTINCT

- Sometimes a query will return a result-set with duplicate tuples. We can remove these duplicate tuples by using the DISTINCT keyword.

```
SELECT DISTINCT Country.Name, Capital
FROM Person, Country
WHERE Person.countryID = Country.countryID;
```

Name	Capital
Ireland	Dublin
Italy	Rome
China	Beijing

EXISTS

- The EXISTS operator tests whether or not a table (or result-set) has at least one row (or tuple).
- Example: Which countries have people in the Persons table?

```
SELECT Name FROM Country
WHERE EXISTS (SELECT * FROM Person
              WHERE Country.countryID = Person.countryID);
```

Name
Ireland
Italy
China

- Similar to the example query we used with SELECT DISTINCT. Generally there are more than one way to write a query!

ALIAS

- The ALIAS keyword allows you to give:
 - alternative names to columns;
 - alternative names to tables; and
 - names to different instances of the same table.

```
SELECT P.Name,C.Name AS Nation,Capital
FROM Person AS P,Country AS C
WHERE P.countryID = C.countryID;
```

Name	Nation	Capital
John	Ireland	Dublin
Gina	Italy	Rome
Ping	China	Beijing
Wei	China	Beijing
Ann	Ireland	Dublin

```
SELECT Name,Age
FROM Person AS P1
WHERE NOT EXISTS (SELECT * FROM Person AS P2
                  WHERE P1.Age < P2.Age);
```

Name	Age
Gina	25

- We don't need to use the AS keyword.


```
SELECT Name,Age FROM Person P1
WHERE NOT EXISTS (SELECT * FROM Person P2 WHERE P1.Age < P2.Age);
```

ALL & ANY

- The ALL operator applies a test to all the tuples in a relation and evaluates to *true* if all tests evaluate to *true*.
- Another way to find the eldest person in the Person table.

```
SELECT Name, Age FROM Person
```

```
WHERE Age >= ALL (SELECT Age FROM Person);
```

- The ANY (or SOME) operator applies a test to all the tuples in a relation and evaluates to *true* if any of the tests evaluate to *true*.
- Example: Everyone who is older than someone else in their country.

```
SELECT Name, Age
```

```
FROM Person P1
```

```
WHERE Age > ANY (SELECT Age FROM Person P2  
                WHERE P1.countryID = P2.countryID);
```

Name	Age
John	20
Ping	22

More Nested Queries

- In addition to subqueries (SELECT statements) in the WHERE clause, subqueries can occur in the SELECT and FROM clauses.
 - When a subquery occurs in the WHERE clause, it generates a set of data which we can test against.
 - When a subquery occurs in the FROM clause, it generates the tables against which we will run the queries.
 - When a subquery occurs in the SELECT clause, this subquery will generate part of the result that will be generated by the enclosing query.

More Nested Queries (2)

- Here is a contrived example: The youngest person who is older than someone else in their country!

```
SELECT *
FROM (SELECT Name, Age FROM Person P1
      WHERE Age > ANY (SELECT Age FROM Person P2
                      WHERE P1.countryID = P2.countryID)) Older
WHERE Age <= ALL (SELECT Age FROM Person P1
                 WHERE Age > ANY (SELECT Age FROM Person P2
                                   WHERE P1.countryID = P2.countryID
                                   )
                 );
```

- The subquery in the FROM clause generates a table with attributes Name and Age that has everyone older than someone else in their country. Tables generated in a FROM clause must have an alias.
- The subquery in the WHERE clause generates a list of the ages of people older than someone else in their country.

More Nested Queries (3)

- Subqueries in SELECT clause **must return exactly one** value.
- Example: Query the age of the oldest person in each country.

```
SELECT Name, (SELECT Age FROM Person
              WHERE Person.countryID = Country.countryID AND
                    Age >= ALL (SELECT Age FROM Person
                                WHERE Person.countryID =
                                Country.countryID
                                )
              ) AS MaxAge
FROM Country;
```

Name	MaxAge
Ireland	20
China	22
Italy	25
France	null

More Nested Queries (4)

- Example: Query the name of the oldest person in each country.

```
SELECT Name, (SELECT Name FROM Person
              WHERE Person.countryID = Country.countryID AND
                    Age >= ALL (SELECT Age FROM Person
                              WHERE Person.countryID =
                                Country.countryID
                              )
              ) AS Elder
FROM Country;
```

Name	Elder
Ireland	John
China	Ping
Italy	Gina
France	null

- If we add an additional tuple (6,'Tara',20,1) to the Person relation, what will happen?

UNION

- The UNION operator combines two result-sets from two SELECT statements.
- The SELECT statements to which the UNION operator is applied must have the **same number of columns** with **similar data types**.
- The UNION operator selects distinct values only. To allow duplicate values, use the UNION ALL operator.

- The SQL syntax for the UNION operator is:

```
SELECT column_name(s) FROM table_name(s)
UNION
SELECT column_name(s) FROM table_name(s);
```

- Example:

```
SELECT Person.Name, Age, Country.Name FROM Person, Country
WHERE Person.countryID=Country.countryID AND Country.Name='Ireland'
UNION
SELECT Person.Name, Age, Country.Name FROM Person, Country
WHERE Person.countryID=Country.countryID AND Country.Name='China';
```

JOIN

- The SQL JOIN statement is used to combine data from two or more tables based on a relationship between specific columns in the tables.
- There are several different type of joins:
 - Inner Join** Generates a new table by combining the columns of two tables where the values of the attributes satisfy the join predicate.
 - Left Join** Generate a new table that includes all the rows of the left table and those rows of the right table that satisfy the join predicate. If no matching rows from the right table exists, the null will appear in columns of the right table where there is no match in left table.

JOIN (2)

- Right Join** Generate a new table that includes all the rows of the right table and those rows of the left table that satisfy the join predicate. If no matching rows from the left table exists, the null will appear in columns of the left table where there is no match in right table.
- Full Join** This is a combination of a left join and a right join. The generated table will have null values for every column of the table that lacks a matching row.
- Joins do not add any additional expressivity to SQL. The same results can be achieved using other SQL statements.

(INNER) JOIN

- This is the default join in SQL and returns a result-set that only includes the rows from each table that satisfies the join condition.

- The SQL syntax for the INNER JOIN is:

```
SELECT column_name(s)
```

```
FROM table1 JOIN table2 ON join_condition;
```

```
SELECT *
```

```
FROM Person JOIN Country ON Person.countryID = Country.countryID;
```

ID	Name	Age	countryID	countryID	Name	Capital
1	John	20	1	1	Ireland	Dublin
5	Ann	19	1	1	Ireland	Dublin
3	Ping	22	2	2	China	Beijing
4	Wei	21	2	2	China	Bejing
2	Gina	25	3	3	Italy	Rome

- Using SELECT ... FROM ... WHERE, can you generate the same result-set?

LEFT (OUTER) JOIN

- Let us add the tuple (6,'Pedro',22,5) to the Person relation.
- The left (outer) join in SQL and returns a result-set that only includes the rows from the left table and the rows from the right table that satisfy the join condition. A null will appear in columns of the right table where there is no match in left table.

- The SQL syntax for the LEFT (OUTER) JOIN is:

```
SELECT column_name(s)
```

```
FROM table1 LEFT JOIN table2 ON join_condition;
```

```
SELECT *
```

```
FROM Person LEFT JOIN Country
```

```
ON Person.countryID = Country.countryID;
```

ID	Name	Age	countryID	countryID	Name	Capital
1	John	20	1	1	Ireland	Dublin
2	Gina	25	3	3	Italy	Rome
3	Ping	22	2	2	China	Beijing
4	Wei	21	2	2	China	Bejing
5	Ann	19	1	1	Ireland	Dublin
6	Pedro	22	5	null	null	null

RIGHT (OUTER) JOIN

- The right (outer) join in SQL and returns a result-set that only includes the rows from the right table and the rows from the left table that satisfy the join condition. A null will appear in columns of the left table where there is no match in right table.
- The SQL syntax for the RIGHT (OUTER) JOIN is:

```
SELECT column_name(s)
FROM table1 RIGHT JOIN table2 ON join_condition;
```

```
SELECT *
FROM Person RIGHT JOIN Country
ON Person.countryID = Country.countryID;
```

ID	Name	Age	countryID	countryID	Name	Capital
1	John	20	1	1	Ireland	Dublin
5	Ann	19	1	1	Ireland	Dublin
3	Ping	22	2	2	China	Beijing
4	Wei	21	2	2	China	Beijing
2	Gina	25	3	3	Italy	Rome
null	null	null	null	4	France	Paris

FULL (OUTER) JOIN

- The full (outer) join in SQL is a combination of the left and right joins. A null will appear in columns of the left and right tables where there is no corresponding match in other table.
- The SQL syntax for the FULL (OUTER) JOIN is:

```
SELECT column_name(s)
FROM table1 FULL JOIN table2 ON join_condition;
```

```
SELECT *
FROM Person FULL JOIN Country
ON Person.countryID = Country.countryID;
```

ID	Name	Age	countryID	countryID	Name	Capital
1	John	20	1	1	Ireland	Dublin
2	Gina	25	3	3	Italy	Rome
3	Ping	22	2	2	China	Beijing
4	Wei	21	2	2	China	Beijing
5	Ann	19	1	1	Ireland	Dublin
6	Pedro	22	5	null	null	null
null	null	null	null	4	France	Paris

FULL (OUTER) JOIN (2)

- Not all database systems support full outer joins, e.g. mySQL.
- The following statement simulates the full outer join on the previous slide.

```
SELECT *
FROM Person LEFT JOIN Country
      ON Person.countryID = Country.countryID
UNION ALL
SELECT *
FROM Person RIGHT JOIN Country
      ON Person.countryID = Country.countryID
WHERE Person.countryID IS NULL;
```

- **Aside:** How can you simulate the left outer join and the right outer join in SQL?

NATURAL JOIN

- A NATURAL JOIN is a special type of *equi-join* that joins on attributes (columns) with the same name.
- The attributes joined on are not duplicated.
- The SQL syntax for the NATURAL JOIN is:

```
SELECT column_name(s)
FROM table1 NATURAL JOIN table2;
```

- In our current example, we would get

```
SELECT *
FROM Person NATURAL JOIN Country;
```

Name	countryID	ID	Age	Capital
------	-----------	----	-----	---------

- Why?
- Using NATURAL JOINS is **dangerous** and should be avoided. Adding a column into a table can suddenly change the behaviour of a previously working query.

JOIN ... USING

- A better version of the natural join is the JOIN ... USING statement.
- The USING clause explicitly specifies the attribute on which the natural join is performed on.

```
SELECT *
FROM Person JOIN Country USING (countryID);
```

countryID	ID	Name	Age	Name	Capital
1	1	John	20	Ireland	Dublin
1	5	Ann	19	Ireland	Dublin
2	3	Ping	22	China	Beijing
2	4	Wei	21	China	Beijing
3	2	Gina	25	Italy	Rome

- It is better software engineering practice to use this version of the natural join to avoid accidentally joining on other attributes that may have the same name.
- Generally you use either the USING clause or the ON clause, both not both together (most database systems will not allow both used together).

Aggregate Functions

- Aggregate functions are functions that perform calculations on values from a column and return a single value.
- The common aggregate functions are:

AVG Returns the average value of the data in the column.

COUNT Returns the number of non-null rows in the column.

FIRST Returns the first value in the column.

LAST Returns the last value in the column.

MAX Returns the largest value in the column.

MIN Returns the smallest value in the column.

SUM Returns the sum of all values in the column.

- **COUNT(*)** returns the number of null and non-null rows in the table.

GROUP BY

- The GROUP BY statement is used together with an aggregate function to apply the aggregate function to the values in specified columns.

- The syntax of the GROUP BY statement is:

```
SELECT column_name1 [, column_name2 , ... ],
       aggregate_function(column_name)
FROM table_name
WHERE condition
GROUP BY column_name1 [, column_name2 , ... ]
```

- Example: Average age in each country.

```
SELECT Country.Name, AVG(Person.Age)
FROM Person, Country
WHERE Person.countryID = Country.countryID
GROUP BY Country.Name;
```

Name	AVG(Person.Age)
Ireland	21.5
China	19.5
Italy	25.0

HAVING

- The HAVING clause has the same effect on GROUP BY as the WHERE clause has on SELECT statements. It enables us to specify an inclusion condition for groups.

- The syntax of the HAVING clause is:

```
SELECT column_name1 [, column_name2 , ... ],
       aggregate_function(column_name)
FROM table_name
WHERE condition
GROUP BY column_name1 [, column_name2 , ... ]
HAVING aggregate_function(column_name) operator
       value
```

```
SELECT Country.Name, AVG(Person.Age)
FROM Person, Country
WHERE Person.countryID = Country.countryID
GROUP BY Country.Name
HAVING COUNT(Person.Age) > 1;
```

Name	AVG(Person.Age)
Ireland	21.5
China	19.5

CREATE DATABASE

- The CREATE DATABASE statement is used to create a database that contains relations (tables).
- The SQL syntax for CREATE DATABASE is:
`CREATE DATABASE database_name;`
- Example: To create a database called myFriends.
`CREATE DATABASE myFriends;`
- CREATE SCHEMA is a synonym for CREATE DATABASE.
- The USE statement selects a database on which further SQL commands will operate.
- The SQL syntax for USE is:
`USE database_name;`
- Example: To use a database called myFriends.
`USE myFriends;`

CREATE TABLE

- The CREATE TABLE statement adds a new table (relation) to the current database.
- The SQL syntax for CREATE TABLE is:
`CREATE TABLE table_name
(
 column_name1 data_type [column_constraints],
 column_name2 data_type [column_constraints],
 ...
 column_namen data_type [column_constraints],
 [table_constraint1,
 ...
 table_constraintm]
)`
- The column and table constraints are optional and will be discussed later.

CREATE TABLE (2)

- Example: Creating the Person and Country relations.

```
CREATE TABLE Person (  
  ID INT,  
  Name VARCHAR(45),  
  Age INT,  
  countryID INT  
);
```

```
CREATE TABLE Country (  
  countryID INT,  
  Name VARCHAR(45),  
  Capital VARCHAR(45)  
);
```

- The actual definitions of Person and Country requires constraints on the columns and the tables.

NOT NULL

- NOT NULL is a column constraint that prevents the column from containing any null values.
- The following defines a table where the Name column must have a non-null value.

```
CREATE TABLE Person (  
  ID INT,  
  Name VARCHAR(45) NOT NULL,  
  Age INT,  
  countryID INT  
);
```

UNIQUE

- The UNIQUE constraint identifies a column that **must be unique** in the table.
- Different database systems specify it in different ways; some as a column constraint, others as a table constraint. **mySQL specifies the UNIQUE constraint as a table constraint.**
- The following defines a table where the ID column must be unique.

```
CREATE TABLE Person (  
  ID INT UNIQUE,  
  Name VARCHAR(45) NOT NULL,  
  Age INT,  
  countryID INT  
);
```

DEFAULT

- The DEFAULT constraint is used to assign a default value to a column.
- This value will be assigned to all new records if no other value is specified.
- The following defines a table where the Age column has a default value of 18.

```
CREATE TABLE Person (  
  ID INT UNIQUE,  
  Name VARCHAR(45) NOT NULL,  
  Age INT DEFAULT 18,  
  countryID INT  
);
```

PRIMARY KEY

- The PRIMARY KEY constraint uniquely identifies each record in the table.
- *Primary keys* must contain unique values.
- A *primary key* can be composed of several columns which cannot be null.
- Each table should have one and only one *primary key*.
- The following defines a table where the primary key is the ID column.

```
CREATE TABLE Person (  
  ID INT NOT NULL,  
  Name VARCHAR(45) NOT NULL,  
  Age INT DEFAULT 18,  
  countryID INT,  
  PRIMARY KEY (ID)  
);
```

PRIMARY KEY (2)

- The following defines a table where the primary key is named and is the composition of the ID and Name columns.

```
CREATE TABLE Person (  
  ID INT NOT NULL,  
  Name VARCHAR(45) NOT NULL,  
  Age INT DEFAULT 18,  
  countryID INT,  
  CONSTRAINT pk_Person PRIMARY KEY (ID,Name)  
);
```

FOREIGN KEY

- A FOREIGN KEY is one table must reference a PRIMARY KEY in another table.
- The following defines a table where countryID column is a *foreign key* that references the Country table.

```
CREATE TABLE Person (
  ID INT NOT NULL,
  Name VARCHAR(45) NOT NULL,
  Age INT DEFAULT 18,
  countryID INT,
  PRIMARY KEY (ID),
  FOREIGN KEY (countryID) REFERENCES
  Country(countryID)
);
```

- The FOREIGN KEY constraint prevents invalid data being inserted into a *foreign key* column because **it has to be one of the values contained in the table it references.**

CHECK

- The CHECK constraint limits the values that can be inserted into the columns of a table.
- **If the CHECK constraint is defined as a column constraint it only allows certain values to be inserted into this column.**
- **If the CHECK constraint is defined as a table constraint it can use the values in other columns to limit the values to be inserted into specific columns.**
- The following defines a table where Age column must have values in the range [1...100].

```
CREATE TABLE Person (
  ID INT NOT NULL,
  Name VARCHAR(45) NOT NULL,
  Age INT DEFAULT 18,
  countryID INT,
  CHECK (Age>0 AND Age <=100),
  PRIMARY KEY (ID),
  FOREIGN KEY (countryID) REFERENCES Country(countryID)
);
```

Referential Triggers

- *Referential Integrity* means that the *foreign key* in each row in the referencing table must match a *primary key* in the referenced table.
- When records are added, modified or deleted from a table *referential integrity* may be violated. SQL has *referential trigger actions* that are activated when a referenced table is modified.
- The triggers can be invoked ON DELETE or ON UPDATE and can have the following options:
 - CASCADE** Whenever rows in the referenced table are modified, the corresponding rows (with matching foreign keys) in the referencing table will get deleted or modified as well.

Referential Triggers (2)

SET NULL Whenever rows in the referenced table are modified, the corresponding foreign keys in the referencing table are set to null.


SET DEFAULT Whenever rows in the referenced table are modified, the corresponding foreign keys in the referencing table are set to the default value.

```
CREATE TABLE Person (  
  ID INT NOT NULL,  
  Name VARCHAR(45) NOT NULL,  
  Age INT DEFAULT 18,  
  countryID INT NOT NULL DEFAULT 1,  
  CHECK (Age>0 AND Age <=100),  
  PRIMARY KEY (ID),  
  FOREIGN KEY (countryID) REFERENCES Country(countryID)  
  ON DELETE SET DEFAULT ON UPDATE CASCADE  
);
```


CREATE INDEX

- The CREATE INDEX statement is used to create indices in tables.
- Indices **enable faster searches but slows down updates**. Should only create indices on columns that will be frequently searched.
- The SQL syntax for CREATE INDEX with duplicate values is:
`CREATE INDEX index_name
ON table_name (column_name);`
- The SQL syntax for CREATE INDEX with no duplicate values allowed:
`CREATE UNIQUE INDEX index_name
ON table_name (column_name);`

ALTER TABLE

- The ALTER TABLE statement is used to add, delete or modify columns in a table.
- To add a column to a table use the following syntax:
`ALTER TABLE table_name
ADD column_name datatype;`
- To delete a column from a table use the following syntax:
`ALTER TABLE table_name
DROP COLUMN column_name;`
- To modify a column in a table use the following syntax:
`ALTER TABLE table_name
MODIFY column_name datatype;`
- Example:
`ALTER TABLE Person
MODIFY Age FLOAT DEFAULT 18.0;` 

DROP

- Tables, indices and databases can be deleted using the DROP statement.
- To delete a table use:
`DROP TABLE table_name;`
- To delete an index use:
`DROP INDEX index_name ON table_name;`
- To delete a database use:
`DROP DATABASE database_name;`

INSERT INTO

- To add a tuple to an existing table use the INSERT INTO statement.
- The SQL syntax for the INSERT INTO statement is:
`INSERT INTO table_name (column1,column2,...) VALUES (value1,value2,...);`
- Example: To insert (6,'Pedro',22,5) into the Person table:
`INSERT INTO Person (ID,Name,Age,countryID) VALUES (6, 'Pedro', 22,5);`
- What effect would the following statement have to the Person table given its definition on the slide titled CHECK?
`INSERT INTO Person (ID,Name,Age,countryID) VALUES (6, 'Pedro', 22,5);`

UPDATE

- The UPDATE statement modifies existing tuples in a table.
- The SQL syntax for the UPDATE statement is:
`UPDATE table_name`
`SET column1=value1, column2=value2,...`
`WHERE condition;`
- **Note:** The WHERE clause in the UPDATE statement specifies which tuples in the table are modified. If there is no WHERE clause **all** tuples are modified!
- Example: If we want to change Ann's age and country to 26 and Italy respectively, then:
`UPDATE Person`
`SET Age=26, countryID=3`
`WHERE Name='Ann' ;`

DELETE FROM

- The UPDATE statement deletes existing tuples from a table.
- The SQL syntax for the UPDATE statement is:
`DELETE FROM table_name`
`WHERE condition;`
- Example: To delete everyone less than 20, then:
`DELETE FROM Person`
`WHERE Age < 20;`
- To delete all tuples from a table use
`DELETE * FROM table_name;`
or `DELETE FROM table_name;`

Views

- A *View* is a named derived table and has two forms:
 - a virtual view; and
 - a materialized view.
- Views are used to:
 - hide data from some users;
 - to modularise the database; and
 - to make some queries more natural or easier.
- *Virtual views* are not actually stored in the database. They are a window onto the base tables that actually store the data. When you query a virtual view the data in the view is dynamically derived from the base tables and other views.
- *Materialized views* are **actual** tables derived from the base tables and other views. Materialised views can make some queries more efficient, but at a cost as modifications to the view and underlying base tables can be costly.

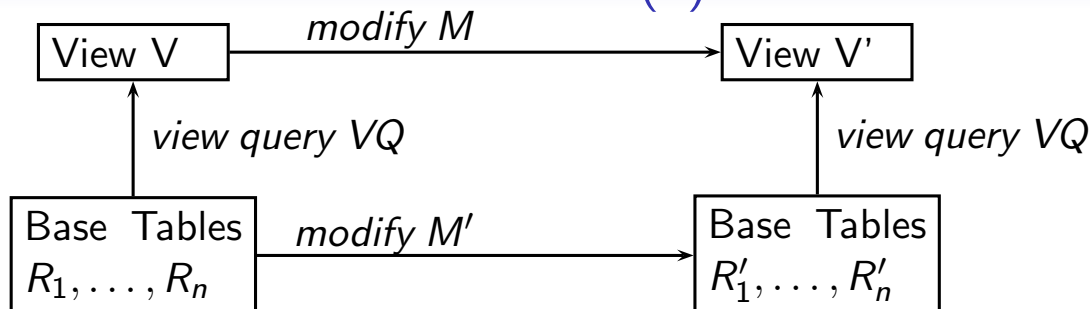
Views (2)

- Views are based on the 3 layer model of a database.
 - Physical** These are the actual pages of the physical disk in which the data is stored.
 - Logical** These are the SQL relations that organise the data. It is an abstraction of the physical data.
 - Conceptual** These are the views that are defined as queries over relations and other views. They are a further level of abstraction.
- Real database applications tend to use **lots and lots** of views.
- We will focus on *virtual views*.

View Definition

- The SQL syntax for defining a view is:
`CREATE VIEW view_name [column_name1, column_name2,...]
 AS subquery
 [WITH CHECK OPTION];`
- If the view is not defined with its own column names, it will inherit the column names from the subquery.
- The subquery of a view cannot use UNION or ORDER BY statements.
- To delete a view use:
`DROP VIEW view_name;`
- Depending on the DBMS you are using, dropping a view may cause other views defined using this view to be dropped as well.

Views (3)



- The view V is defined as a query VQ on a set of base tables R_1, \dots, R_n .
- If the view is modified, then the base tables would need to be modified to R'_1, \dots, R'_n so that the query VQ results in the view V' .
- Working out the modification M' from M is not straight forward.
 - The view designer can define for each view how it is modified.
 - M' can be derived automatically if restrictions are placed on the view definition (SQL standard).

View (4)

- For a view to be automatically updatable:
 - SELECT (no DISTINCT) on a single table T .
 - Attributes not in the view can be null or have a default value.
 - Subqueries must not refer to T .
 - No GROUP BY or aggregation.
- Consider the following definition of Person:

```
CREATE TABLE Person (  
  ID INT NOT NULL,  
  Name VARCHAR(45) NOT NULL,  
  Age INT DEFAULT 18,  
  countryID INT NOT NULL DEFAULT 1,  
  PRIMARY KEY (ID),  
  FOREIGN KEY (countryID) REFERENCES Country(countryID),  
);
```
- and the following view:

```
CREATE VIEW Chinese(number,name)  
AS SELECT ID,Name  
FROM Person  
WHERE countryID = 2;
```

View (5)

- Even though the view Chinese follows the restrictions for an updatable table, consider the following update:

```
INSERT INTO Chinese (number,name)  
VALUES (7,'David');
```
- This update to the view will result in an update in the base tables. In this case a tuple (7,'David',18,1) into the Person table.
- However the view Chinese will not change, so should have the update of the base table been allowed? Probably not.

View (6)

- The update can be prevented by defining the view with the WITH CHECK OPTION as follows:

```
CREATE VIEW Chinese(number,name)
AS SELECT ID,Name
FROM Person
WHERE countryID = 2
WITH CHECK OPTION;
```

- The WITH CHECK OPTION prevents updates that do not change the view.

Views (7)

- If the view does not need to be automatically modifiable then the restrictions mentioned on the earlier slide do not apply.

```
CREATE VIEW WhoWhere(number,name,country)
AS SELECT ID,Person.Name,Country.Name
FROM Person JOIN Country USING(countryID);
```

- When a query is run on the view WhoWhere the query is rewritten inside the DBMS in terms of the base tables.

```
SELECT number,name
FROM WhoWhere
WHERE country='Ireland';
```

could be changed internally to:

```
SELECT ID,Person.Name
FROM Person JOIN Country USING (countryID)
WHERE Country.Name='Ireland';
```

Views (8)

- Even though the SQL standard has strong restrictions on what can be an “updatable view”:
 - SELECT (no DISTINCT) on a single table T .
 - Attributes not in the view can be null or have a default value
 - Subqueries must not refer to T .
 - No GROUP BY or aggregation.

several DBMS relax these restrictions.

- For example, mySQL will allow the following and implement it correctly.
- Consider the following view definition:

```
CREATE VIEW Old(ID,Name,Country)
AS SELECT ID,Person.Name,Country.Name
FROM Person,Country
WHERE Person.countryID = Country.countryID AND
      Age > 20
WITH CHECK OPTION;
```

Views (9)

- `SELECT * FROM Old;` produces:

ID	Name	Country
3	Ping	China
4	Wei	China
2	Gina	Italy

- Strictly, according to the SQL standard, the view Old should not be updatable.
- But,

```
UPDATE Old SET Name='Lihwa' WHERE ID=3;
```

 will work correctly.
- `SELECT * FROM Old;` now produces:

ID	Name	Country
3	Lihwa	China
4	Wei	China
2	Gina	Italy

and modifies the underlying Person table correctly.

Materialized Views

- *Materialized views* differ from *virtual views* in that a table representing the view is actually created.
- In addition to the advantages of *virtual views*, *materialized views* can offer a considerable speed improvement during searches.
- Advanced DBMS will try to figure out how to best use existing *materialized views* when performing a query.
- An important issue relating to *materialized views* is keeping the *materialized view* in synchronisation with the base tables it references.
 - Updates become slower as changes in the base tables may require an incremental or complete rebuilding of the *materialized view*.
 - Updates to the *materialized view* may require the base tables to be modified.
 - This is the classic Efficiency-Update trade-off!

Materialized Views (2)

- The SQL syntax for a MATERIALIZED VIEW is:
CREATE MATERIALIZED VIEW *view_name*
AS *subquery*;
- *Materialized views* are used for:
 - denormalisation;
 - replication;
 - data warehousing; and
 - validation.
- In addition to the disadvantage of maintaining synchronisation between the *materialized views* and the base tables, *materialized views* also can use lots of data space.