SQL

Fabrice Rossi

CEREMADE
Université Paris Dauphine

2020

SQL

What is SQL?

- SQL is relational data management language
- Structured Query Language pronounced sequel
- developed initially by IBM as an "implementation" of the relational model
- SQL is a standard since 1986 (numerous versions)

Implementations

- SQL is “supported” by all relational database management systems
- many open source solutions (MySQL/MariaDB, PostgreSQL, SQLite, etc.)
- but many variations in the support level (portability is not guaranteed)

SQL Components

Multiple aspects

- Data Definition/Description Language
  - relational model description
  - domain definition
- Data Manipulation Language: insertion, suppression and modification
- Data Query Language
  - read only manipulation
  - selection, filtering, grouping, etc.
- Data Control Language
  - access control to the databases
  - users, roles, permissions, etc.

Outline

Data Description Language

Data Query Language

Data Manipulation Language

SQL Data Description Language

Relation

- a relation is created in SQL by

  `CREATE TABLE relation_name (column_name domain, ...);`
- SQL supports numerous default domains (implementation dependent!):
  - exact numeric values
    - `INT`, `SMALLINT`, `BIGINT`
    - `NUMERIC(p,s)` and `DECIMAL(p,s)`
  - approximate numeric: `FLOAT`, `DOUBLE`
  - `DATETIME`, `DATE` and `TIME`: date and time
  - `BOOLEAN`: true or false
  - `CHAR(n)` and `VARCHAR(n)`: string with maximum size n
- implementation specific extensions

Example

IMDB database

Actors

id	first_name	last_name	gender	film_count
567368	Olivia	Burnette	F	1
758314	Beata	Pozniak	F	1
636385	Joanne	Gordon	F	1
588101	Suzanne	Cox	F	1
683913	Melissa	Kurtz	F	1

$$\begin{aligned}&\mathit{Actors}(\mathit{id} : \mathbb{N}^{+}, \mathit{first\_name} : \mathit{string},\\ &\qquad \mathit{last\_name} : \mathit{string},\\ &\qquad \mathit{gender} : \{F, M\}, \mathit{film\_count} : \mathbb{N}^{+})\end{aligned}$$

```
CREATE TABLE Actors (
  id INT, first_name VARCHAR(100),
  last_name VARCHAR(100),  gender CHAR(1),
  film_count INT
);
```

Integrity constraints

Somme constraints

- **PRIMARY KEY**: self explanatory
- **FOREIGN KEY**: self explanatory
- **UNIQUE**: candidate key
- **NOT NULL**: non nullable

Example

Actors(id, first_name, last_name, gender, film_count)

```
CREATE TABLE Actors (
  id INT PRIMARY KEY, first_name VARCHAR(100),
  last_name VARCHAR(100), gender CHAR(1),
  film_count INT
);
```

Integrity constraints

Somme constraints

- **PRIMARY KEY**: self explanatory
- **FOREIGN KEY**: self explanatory
- **UNIQUE**: candidate key
- **NOT NULL**: non nullable

Example

Actors(id, first_name, last_name, gender, film_count)

```
CREATE TABLE Actors (
  id INT, first_name VARCHAR(100),
  last_name VARCHAR(100), gender CHAR(1),
  film_count INT,
  PRIMARY KEY (id)
);
```

Integrity constraints

Somme constraints

- **PRIMARY KEY**: self explanatory
- **FOREIGN KEY**: self explanatory
- **UNIQUE**: candidate key
- **NOT NULL**: non nullable

Example

Actors(id, first_name, last_name, gender, film_count)

```sql
CREATE TABLE Actors (
  id INT PRIMARY KEY,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  gender CHAR(1) NOT NULL,
  film_count INT NOT NULL
);
```

Keys

Primary keys

- primary keys are not mandatory in SQL
- but they should be specified!
- **`UNIQUE`** is useful as a constraint
- a primary key can be made with several columns using
  `PRIMARY KEY (COL1, COL2, ...)`
  in the table creation

Foreign keys

- declared as `FOREIGN KEY (column)` during table creation
- together with a `REFERENCES table(column)`
- a foreign key can be a set of columns

Example

IMDB database simplified

- Actors(id, first_name, last_name)
- Movies(id, name)
- Roles(#actor_id,#movie_id,role)

```
CREATE TABLE Actors(id INT PRIMARY KEY,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL);
CREATE TABLE Movies(id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL);
CREATE TABLE Roles(actor_id INT, movie_id INT,
   ROLE VARCHAR(100) NOT NULL,
   PRIMARY KEY (actor_id, movie_id),
   FOREIGN KEY (actor_id) REFERENCES Actors(id),
   FOREIGN KEY (movie_id) REFERENCES Movies(id));
```

Referential integrity

Foreign keys

- must reference an existing primary key
- SQL allows one to handle consequences of tuple modifications
  - what happens if the primary key of a tuple is modified?
  - what happens if a tuple is deleted?
- **`ON DELETE`** `something` and **`ON UPDATE`** `something`
- with something being
  - **`CASCADE`**: propagate the modification to referring tuples
  - **`RESTRICT`**: forbid the modification if there are referring tuples
  - **`SET NULL`** or **`SET DEFAULT`**: modify the foreign key in the referring tuples as described

Example

IMDB database simplified

- Actors(id, first_name, last_name)
- Movies(id, name)
- Roles(#actor_id,#movie_id,role)

```
CREATE TABLE Roles(actor_id INT, movie_id INT,
   ROLE VARCHAR(100) NOT NULL,
   PRIMARY KEY (actor_id, movie_id),
   FOREIGN KEY (actor_id) REFERENCES Actors(id)
      ON DELETE RESTRICT ON UPDATE CASCADE,
   FOREIGN KEY (movie_id) REFERENCES Movies(id)
      ON DELETE RESTRICT ON UPDATE CASCADE);
```

Domains

SQL domains

- domains can be created in SQL
- typical form

```
CREATE DOMAIN Gender AS CHAR(1)
  CHECK (VALUE IN ('F','M'));
```

- unsupported in many implementations (e.g. MySQL, MariaDB)

Constraints based version

- constraints can be added to the table creation
- `CHECK` can be used to implement domains
- less elegant (no centralized definition)

Example

Actors(id, first_name, last_name, gender, film_count)

```sql
CREATE TABLE Actors (
  id INT, first_name VARCHAR(100),
  last_name VARCHAR(100), gender CHAR(1),
  film_count INT,
  PRIMARY KEY (id),
  CONSTRAINT gender_check CHECK(gender in ('F','M'))
);
```

Additional commands

Modifying the model

- `DROP TABLE name;`: deletes a table
- `DELETE FROM name;`: empties a table
- `ALTER TABLE ...;`: schema modification
  - add an attribute:
    `ALTER TABLE name ADD attribute domain;`
  - remove an attribute: `ALTER TABLE name DROP attribute;`
  - changing the properties of a column (domain, constraints, etc.)
  - etc.

Outline

Data Description Language

Data Query Language

Data Manipulation Language

SQL Data Query Language

The **SELECT** command

- the main query command in SQL
- general form

```
SELECT something FROM somewhere
[WHERE conditions] [GROUP BY grouping]
[HAVING group conditions] [ORDER BY something]
```

- provides all the manipulations available in the relational algebra:
  - subsetting, filtering, transforming
  - summarizing
  - joining
- but mainly in a declarative form

Projection

Column oriented subsetting

- simple `SELECT` queries can be used to subset a relation on interesting attributes
- general form `SELECT a1, ..., aN FROM relation;`

Example $\Pi_{name}(Movies)$

```
SELECT name FROM Movies;
```

Movies

id	name	year	rank
10920	Aliens	1986	8.20
17173	Animal House	1978	7.50
18979	Apollo 13	1995	7.50
30959	Batman Begins	2005	0.00
46169	Braveheart	1995	8.30

Result

name
Aliens
Animal House
Apollo 13
Batman Begins
Braveheart

Transformation

Expression and renaming

- columns may be renamed using `orig_name AS new_name` in the **`SELECT`** command
- simple calculations may also be performed on columns including the results as new columns

Example $\Pi_{Title=name,Note=rank+1}(Movies)$

```
SELECT name as Title, rank+1 as Note FROM Movies;
```

Movies

id	name	year	rank
10920	Aliens	1986	8.20
17173	Animal House	1978	7.50
18979	Apollo 13	1995	7.50
30959	Batman Begins	2005	0.00
46169	Braveheart	1995	8.30

Result

Title	Note
Aliens	9.20
Animal House	8.50
Apollo 13	8.50
Batman Begins	1.00
Braveheart	9.30

Selection

Selecting tuples

- the **WHERE** clause can be used to select tuples fulfilling some conditions
- general form

```
SELECT columns FROM table WHERE conditions;
```

Example $\Pi_{Title=name,Note=rank+1}(\sigma_{year=2000}(Movies))$

```
SELECT name as Title, rank as Note FROM Movies
WHERE year=2000;
```

Movies

id	name	year	rank
10920	Aliens	1986	8.20
17173	Animal House	1978	7.50
18979	Apollo 13	1995	7.50
30959	Batman Begins	2005	0.00
46169	Braveheart	1995	8.30

Result

Title	Note
Hollow Man	5.30
Memento	8.70
O Brother, Where Art Thou?	7.80
Snatch.	7.90

Cartesian product

Multiple relations

- **SELECT** queries can operate on several relations
- general from

  ```
  SELECT a_1, ..., a_N FROM r_1, ..., r_P WHERE cond;
  ```
- cartesian product semantics

$$\Pi_{a_1,\ldots,a_N}(\sigma_{cond}(r_1 \times \ldots \times r_P))$$

- explicit particular cases (such as natural join)
- notice that renaming of the relations with **AS** is possible and simplifies writing the conditions

Example

IMDB database

```
SELECT last_name, role, name AS title
 FROM Actors AS A, Movies AS M, Roles AS R
 WHERE A.id = R.actor_id AND R.movie_id = M.id;
```

last_name	role	title
Armstrong	Lydecker	Aliens
Benedict	Russ Jorden	Aliens
Biehn	Cpl. Dwayne Hicks	Aliens
Fairman	Doctor	Aliens
Henn	Timmy Jorden	Aliens
Henriksen	Bishop	Aliens
Hope	Lt. Gorman	Aliens
Kash	Pvt. Spunkmeyer	Aliens
Lees	Power Loader Operator	Aliens
Matthews	Sgt. Apone	Aliens

$$\Pi_{last\_name, role, title = name}(Actors \bowtie_{id = actors\_id} Roles \bowtie_{movie\_id = id} Movies)$$

Explicit joins

More declarative queries

- general form

  `SELECT ... FROM r1 something JOIN r2 ON condition;`
- type of join (something)
  - `INNER JOIN`
  - `LEFT [OUTER] JOIN` and `RIGHT [OUTER] JOIN`
  - `FULL [OUTER] JOIN`
  - `NATURAL JOIN`
- `CROSS JOIN` can be used for cartesian product but does not support `ON`

Example

Implicit

```
SELECT last_name, role, name AS title
 FROM Actors, Movies, Roles
 WHERE Actors.id = Roles.actor_id AND Roles.movie_id = Movies.id;
```

Explicit

```
SELECT last_name, role, name AS title
 FROM Actors INNER JOIN Roles ON Actors.id = Roles.actor_id
      INNER JOIN Movies on Roles.movie_id = Movies.id;
```

Subtleties

WHERE versus ON

- more general form

  ```
  SELECT ... FROM r1 something JOIN r2 ON cond1 WHERE cond2;
  ```

  - `cond1` applies *during* the join operation
  - `cond2` applies to the resulting relation
- compared to

  ```
  SELECT ... FROM r1, r2 WHERE cond1 AND cond2;
  ```

  - we start with $r_1 \times r_2$
  - `cond1 AND cond2` apply on the cartesian product
  - no **`NULL`** completion!
- only affects outer joins

Example

RA

id	txt
1	first
2	second

RB

id	ref
1	1
2	2
3	NULL

```
SELECT * from FROM
     RB INNER JOIN RA
     ON RB.ref=RA.id;
```

id	ref	txt
1	1	first
2	2	second

Example

RA

id	txt
1	first
2	second

RB

id	ref
1	1
2	2
3	NULL

```
SELECT * FROM
      RB, RA
      WHERE RB.ref=RA.id;
```

id	ref	txt
1	1	first
2	2	second

Example

RA

id	txt
1	first
2	second

RB

id	ref
1	1
2	2
3	NULL

```
SELECT * FROM
    RB LEFT OUTER JOIN RA
    ON RB.ref=RA.id;
```

id	ref	txt
1	1	first
2	2	second
3	NULL	NULL

Example

RA

id	txt
1	first
2	second

RB

id	ref
1	1
2	2
3	NULL

```
SELECT * FROM
    RB LEFT OUTER JOIN RA
    ON RB.ref=RA.id
    WHERE RB.ref is NULL;
```

id	ref	txt
3	NULL	NULL

Example

RA

id	txt
1	first
2	second

RB

id	ref
1	1
2	2
3	NULL

```
SELECT * FROM
    RB, RA
    WHERE RB.ref=RA.id
    AND RB.ref is NULL;
```

id	ref	txt

Aggregation

Global summaries

- aggregation functions can be used in the result part of the **`SELECT`** command
- they operate at the column level
- some examples:
  - **`COUNT`** and **`COUNT`**`(`**`DISTINCT`**`(.))`
  - **`MAX`**, **`MIN`**, **`SUM`**
  - **`AVG`**, `STD`, `VARIANCE`

Financial database

```
SELECT COUNT(*) FROM Actors WHERE Gender='F';
```

count(*)
443

Conditional aggregation

Grouped aggregation in SQL

- the **GROUP BY** clause of the **SELECT** command provides conditional analysis
- it *splits* the relation into groups of tuples on which it *applies* chosen aggregation functions
- groups can be further selected based on global properties with the **HAVING** clause

General form

```
SELECT aggregates FROM relation
       [WHERE conditions]
       GROUP BY columns
       [HAVING group conditions]
```

Examples

Count actors per gender

```
SELECT gender, COUNT(*) AS number
   FROM Roles
   GROUP BY gender;
```

gender	number
M	1464
F	443

Average rank per year

```
SELECT year, AVG(rank) AS avg_rank
    FROM Movies
    GROUP BY year;
```

year	avg_rank
1972	9.00
1977	8.80
1978	7.50
1984	5.80
1986	8.20
1987	7.20
1989	6.95

Group selection

Having

- the **HAVING** clause selects only certain groups
- groups are selected based on a predicate which can use group aggregation
- the **SELECT** part applies to selected groups

Example

```
SELECT year,
    AVG(rank) AS avg_rank
    FROM Movies GROUP BY year
    HAVING AVG(rank)>=8;
```

year	avg_rank
1972	9.00
1977	8.80
1986	8.20
1994	8.85
1996	8.20
2004	8.25

Example

Aggregation and join

- Genre relation in IMDB database
- Genre(movie_id: $\mathbb{N}^+$, genre: string)

movie_id	genre
10920	Action
10920	Horror
10920	Sci-Fi
10920	Thriller
17173	Comedy

Example

Aggregation and join

- Genre relation in IMDB database
- Genre(movie_id: $\mathbb{N}^+$, genre: string)

movie_id	genre
10920	Action
10920	Horror
10920	Sci-Fi
10920	Thriller
17173	Comedy

```
SELECT genre, COUNT(*) AS count
       FROM Movies LEFT JOIN Genres ON id=movie_id
       GROUP BY genre;
```

genre	count
Action	8
Adventure	5
Animation	2
Comedy	11
Crime	12

Example

```
SELECT first_name, last_name, COUNT(DISTINCT(genre)) as genres
   FROM Actors INNER JOIN Roles ON Actors.id = Roles.actor_id
               INNER JOIN Movies ON Roles.movie_id = Movies.id
               INNER JOIN Genres ON Movies.id = Genres.movie_id
   GROUP BY first_name, last_name;
```

Example

```
SELECT first_name, last_name, COUNT(DISTINCT(genre)) as genres
   FROM Actors INNER JOIN Roles ON Actors.id = Roles.actor_id
               INNER JOIN Movies ON Roles.movie_id = Movies.id
               INNER JOIN Genres ON Movies.id = Genres.movie_id
   GROUP BY first_name, last_name;
```

first_name	last_name	genres
'Weird Al'	Yankovic	1
A. Ray	Ratliff	3
Aaron	Sorkin	2
Aaron James	Cash	2
Abdul	Blackmanwest	5
Abe	Vigoda	2
Abraham	Aronofsky	2
Ada	Nicodemou	3
Adam	Fogerty	2
Adam	LeGrant	5

Sorting the results

```
SELECT ... ORDER BY A1, ..., AK;
```

- sorting the result using the specified attributes
- lexicographic ordering
- `DESC` and `ASC` specify the sorting order

Sorting the results

```
SELECT ... ORDER BY A1, ..., AK;
```

- sorting the result using the specified attributes
- lexicographic ordering
- **DESC** and **ASC** specify the sorting order

```
SELECT genre, COUNT(*) AS count
       FROM Movies LEFT JOIN Genres ON id=movie_id
       GROUP BY genre
       ORDER BY count DESC;
```

genre	count
Drama	17
Thriller	17
Crime	12
Comedy	11
Action	8

Combining relations

Set operations

- results of **`SELECT`** queries can be combined
- three standard operations: **`UNION`**, **`INTERSECT`** and **`EXCEPT`**
- standard use: no duplicates
- multi set version: add the **`ALL`** keyword after the operation to keep duplicates

Example

IMDB database

- Directors relation
- Directors(id, first_name, last_name)

All persons

```
(SELECT first_name, last_name FROM Actors)
UNION
(SELECT first_name, last_name FROM Directors)
ORDER BY last_name, first_name;
```

first_name	last_name
Pamela	Abdy
Lewis	Abernathy
Andrew	Adamson
William	Addy
Kelly	Adkins

Example

IMDB database

- Directors relation
- Directors(id, first_name, last_name)

All persons

```
(SELECT 'Actor' as role, first_name, last_name FROM Actors)
UNION
(SELECT 'Director' as role, first_name, last_name FROM Directors)
ORDER BY last_name, first_name;
```

role	first_name	last_name
Director	Pamela	Abdy
Director	Lewis	Abernathy
Actor	Andrew	Adamson
Director	Andrew	Adamson
Director	William	Addy

Subqueries

Principle

- **SELECT** queries can be used as parts of other **SELECT** queries
- *nested* subqueries
- typical uses
  - complex conditions in the **WHERE** clause
  - new relation in the **FROM** clause
  - attributes computed by a query

Example

Above average movies

- aggregates cannot be used in a **WHERE** clause

```
-- this is incorrect
SELECT * FROM movies WHERE rank > AVG(rank);
```

- use a subquery in the **WHERE** clause

```
SELECT * FROM movies
      WHERE rank > (SELECT AVG(rank) FROM movies)
      ORDER BY rank DESC;
```

id	name	year	rank
130128	Godfather, The	1972	9.00
297838	Shawshank Redemption, The	1994	9.00
313459	Star Wars	1977	8.80
210511	Memento	2000	8.70
267038	Pulp Fiction	1994	8.70

Example

Number of roles in each movie

Join based solution

```
SELECT id, name, year, rank, COUNT(role) AS num_role
      FROM movies INNER JOIN roles
           ON roles.movie_id = movies.id
      GROUP BY id, name, year, rank;
```

Example

Number of roles in each movie

With a (correlated) subquery

```
SELECT *, (SELECT COUNT(*)
          FROM roles WHERE roles.movie_id=movies.id
          ) AS num_role
     FROM movies;
```

Example

Number of roles in each movie

With a (correlated) subquery

```
SELECT *, (SELECT COUNT(*)
          FROM roles WHERE roles.movie_id=movies.id
          ) AS num_role
      FROM movies;
```

Warning

In general, joins are more efficient than subqueries (especially for correlated subqueries)

Set membership

Testing for set membership

- a subquery returns a relation (a (multi)-set)
- the [NOT] IN operator can be used in a WHERE clause to check whether a tuple is in the corresponding relation

Actors without no “homonym” in the directors relation

```
SELECT * FROM Actors
         WHERE (first_name, last_name) NOT IN
          (SELECT first_name, last_name FROM Directors);
```

Set operations in where clauses

More set oriented operations

WHERE clause can also

- test for emptiness with [**NOT**] **EXISTS**
- test for uniqueness with [**NOT**] **UNIQUE** (seldom supported)
- compare numerical sets with **SOME** and **ALL**

Rank conditions in movies

- better than at least a movie from 1995

```
SELECT * FROM Movies
         WHERE rank > SOME
           (SELECT rank FROM Movies WHERE year = 1995);
```

- better than all movies from 1995

```
SELECT * FROM Movies
         WHERE rank > ALL
           (SELECT rank FROM Movies WHERE year = 1995);
```

Examples

Actors who are directors

```
SELECT * FROM Actors as A
         WHERE EXISTS
          (SELECT * FROM Directors as D
                   WHERE A.first_name=D.first_name
                         AND A.last_name=D.last_name);
```

Examples

Actors who are directors

```
SELECT * FROM Actors
         WHERE (first_name, last_name) NOT IN
          (SELECT first_name, last_name FROM Directors);
```

Examples

Actors who are directors

```
SELECT * FROM Actors
         WHERE (first_name, last_name) NOT IN
          (SELECT first_name, last_name FROM Directors);
```

Actors who played only one role

```
SELECT * FROM actors
         WHERE UNIQUE
          (SELECT * FROM roles WHERE actors.id = actor_id);
```

Examples

Actors who are directors

```
SELECT * FROM Actors
         WHERE (first_name, last_name) NOT IN
          (SELECT first_name, last_name FROM Directors);
```

Actors who played only one role

```
SELECT * FROM actors
         WHERE
          (SELECT COUNT(*) FROM roles WHERE actors.id = actor_id)=1;
```

Examples

Actors who are directors

```
SELECT * FROM Actors
         WHERE (first_name, last_name) NOT IN
          (SELECT first_name, last_name FROM Directors);
```

Actors who played only one role

```
SELECT actors.* FROM actors INNER JOIN
       (SELECT actor_id FROM roles GROUP BY actor_id
                                   HAVING count(*)=1)
       AS unique_role
       ON id=actor_id;
```

Examples

Actors who are directors

```
SELECT * FROM Actors
         WHERE (first_name, last_name) NOT IN
          (SELECT first_name, last_name FROM Directors);
```

Actors who played only one role

```
SELECT actors.* FROM actors INNER JOIN roles
        ON id=actor_id
        GROUP BY id, first_name, last_name, gender, film_count,
                 actor_id
        HAVING count(*)=1;
```

Outline

Data Description Language

Data Query Language

Data Manipulation Language

SQL Data Manipulation Language

INSERT

- inserting a tuple into a relation:
  `INSERT INTO table VALUES (...);`
- variants include
  `INSERT INTO table (columns...) VALUES (...);` to specify the column names (`NULL` is assigned to missing columns)

DELETE

- deleting is done conditionally, using a `WHERE` clause
- general syntax
  `DELETE FROM table WHERE condition;`

Updating

UPDATE

- used to alter tuples
- general syntax

```
UPDATE table
  SET column = value [,column = value...]
  [WHERE condition];
```

Changelog

- November 2020: initial version

Licence

Version

Last git commit: 2020-11-23
By: Fabrice Rossi (Fabrice.Rossi@apiacoa.org)
Git hash: 312a0636ceb585db2da88a95e73b59651b34a3fb