SQL crash course

Louis Jachiet

SQL Motivation

The problem with programming languages

Express what you want and not how to get it

I want the directors of movies with "Greta Gerwig" as actress

VS

List movies, if "Greta Gerwig" appears in the list of actors, output movie director.

The problem with programming languages

Abstract away the way the data is stored

- logical representation of data
- easy to update the representation
- easy to add features (persistence, concurrency, etc.)
- easy to add optimization

Before database systems

Each application maintaining data would have to deal with:

- Structure
- Persistence
- Efficiency
- Update without breaking constraints
- Concurrency
- ...

The first database systems

In the first database systems the application would access the data through an API.

Typically like a key-value store

The first database systems

In the first database systems the application would access the data through an API.

Typically like a key-value store

Structure ∼OK

Persistence OK

Efficiency NO

Update without breaking constraints meh

Concurrency meh

...

Arrival of SQL

In 1970, Ted Codd proposes the Relational Model and Relational Algebra.

In Ted proposal the user of a database only specifies what data it wants and not how to get it.

Generalities

The *Structured Query Language* (SQL) was introduced in 1974 after the work of Ted Codd.

Generalities

The *Structured Query Language* (SQL) was introduced in 1974 after the work of Ted Codd.

It became an official standard in 1986

new version of the standard in 89, 92, 99, 03, 08, 11, etc.

Generalities

The *Structured Query Language* (SQL) was introduced in 1974 after the work of Ted Codd.

It became an official standard in 1986

new version of the standard in 89, 92, 99, 03, 08, 11, etc.

Very well supported *with some variations...*

Oracle, DB2, SQL Server, SQLlite, Postgres, MySQL/MariaDB

Data model

An example

Theaters		
Name	Address	nbRooms
"La Nef"	"bd Édouard Rey"	7
"Le Mélies"	"caserne de Bonne"	3
"Le Club"	"rue Phalanstère"	3

Casting		
Movie	Person	Role
"Inception"	"Ellen Page"	Actor
"Inception"	"Leonardo DiCaprio"	Actor
"Inception"	"Christopher Nolan"	Director
"Toy Story 3"	"Tom Hanks"	Voice Actor
"Mamma Mia"	"Meryl Streep"	Actor
"Mamma Mia"	" Phyllida Lloyd"	Director

Projection		
Title	Date	Theater
"Inception"	12/08/2010 20h	"Le Mélies"
"Toy Story 3"	13/08/2010 17h	"Le Club"
"Toy Story 3"	13/08/2010 20h	"Le Club"
"Toy Story 3"	10/08/2010 17h	"Le Mélies"
"Akmareul boatda"	10/08/2010 16h	"Le Club"
"How to train your dragon"	12/03/2010 18h	"Pathé Chavant"

The relational model

A Schema is composed of:

- Several tables or relations.

The relational model

A Schema is composed of:

- Several tables or relations.
- Each relation has several columns or attributes.

The relational model

A Schema is composed of:

- Several tables or relations.
- Each relation has several columns or attributes.
- Each column has a type (`INTEGER`, `BIGINT`, `VARCHAR`, ...)

The data is stored as records or tuples into this table.

An example

Theaters

Name	Address	nbRooms
"La Nef"	"bd Édouard Rey"	7
"Le Mélies"	"caserne de Bonne"	3
"Le Club"	"rue Phalanstère"	3

Casting

Movie	Person	Role
"Inception"	"Ellen Page"	Actor
"Inception"	"Leonardo DiCaprio"	Actor
"Inception"	"Christopher Nolan"	Director
"Toy Story 3"	"Tom Hanks"	Voice Actor
"Mamma Mia"	"Meryl Streep"	Actor
"Mamma Mia"	" Phyllida Lloyd"	Director

Projection

Title	Date	Theater
"Inception"	12/08/2010 20h	"Le Mélies"
"Toy Story 3"	13/08/2010 17h	"Le Club"
"Toy Story 3"	13/08/2010 20h	"Le Club"
"Toy Story 3"	10/08/2010 17h	"Le Mélies"
"Akmareul boatda"	10/08/2010 16h	"Le Club"
"How to train your dragon"	12/03/2010 18h	"Pathé Chavant"

Query

Different types of queries

SQL queries allows to:

- Retrieve data

  SELECT

- Add data

  INSERT

- Delete data

  DELETE

- Update data

  UPDATE

- And many other things (e.g. modify schema)

  ALTER / CREATE TABLE / ...

SELECT queries

SELECT base

```
SELECT col1 as myFancyCol, col2, col3
FROM myTable
```

SELECT base, alternative

```
SELECT *
FROM myTable
```

SELECT base with expression

```
SELECT myCol*3, myCol/someOtherCol, "hello"
FROM myTable
```

SELECT base with condition

```
SELECT *
FROM myTable
WHERE myIntCol > 42
```

SELECT base with condition

```
SELECT *
FROM myTable
WHERE     myIntCol > 42
      AND myStringCol LIKE '%hello%'
```

SELECT base several tables

```
SELECT *
FROM myTable, mySecondTable
```

SELECT base several tables with conditions

```
SELECT *
FROM myTable, mySecondTable
WHERE myTable.someCol = mySecondTable.someCol
```

SELECT base several tables with conditions

```sql
SELECT *
FROM myTable, mySecondTable
WHERE myTable.someCol = mySecondTable.someCol
```

```sql
SELECT *
FROM myTable
INNER JOIN  mySecondTable
ON myTable.someCol = mySecondTable.someCol
```

SELECT base with group

```
SELECT someOtherCol, Max(yetAnotherCol), COUNT(*)
FROM myTable
WHERE myTable.someCol = ``some value''
GROUP BY someOtherCol
```

SELECT base with group

```
SELECT someOtherCol, Max(yetAnotherCol), COUNT(*)
FROM myTable
WHERE myTable.someCol = ``some value''
GROUP BY someOtherCol
```

The “GROUP BY” needs to contain all columns selected!

SELECT base with group

```
SELECT someOtherCol, Max(yetAnotherCol), COUNT(*)
FROM myTable
WHERE myTable.someCol = ``some value''
GROUP BY someOtherCol
```

The "GROUP BY" needs to contain all columns selected!

When aggregates appears on the columns selected an implicit "GROUP BY 1" is added.

Detour: a restricted list of useful aggregates

SUM, AVG, MIN, MAX, STDEV, VAR

COUNT

COUNT DISTINCT

STRING_AGG / GROUP_CONCAT / ...

SELECT base with group and conditions on groups

```
SELECT someOtherCol, max(yetAnotherCol), COUNT(*)
FROM myTable
WHERE myTable.someCol = ``some value''
GROUP BY someOtherCol
HAVING sum(someColInt) > 42
```

SELECT base with order

```
SELECT *
FROM myTable
ORDER BY col, DESC(someCol)
```

SELECT base with limit

```
SELECT *
FROM myTable
ORDER BY col, DESC(someCol)
LIMIT 10
```

SELECT base with limit and offsets

```
SELECT *
FROM myTable
ORDER BY col, DESC(someCol)
LIMIT 10
OFFSET 10
```

General SELECT

```
SELECT cols
FROM tables
WHERE condition
GROUP BY cols2
HAVING condition2
```

Nested SELECT (can be used to replace HAVING)

```
SELECT cols FROM
(
    SELECT *
    FROM tables
    WHERE condition
    GROUP BY cols2
) as t
WHERE condition2
```

A few special SQL constructs

NULL

NULL is a special SQL value to designate a missing value.

NULL

NULL is a special SQL value to designate a missing value.

Because it designates a missing value, it is not equal or comparable to anything.

NULL

NULL is a special SQL value to designate a missing value.

Because it designates a missing value, it is not equal or comparable to anything.

In particular, it will not join with anything

What (SELECT * FROM myTable WHERE ((NULL=NULL) IS NULL) = NULL) returns?

A) myTable

B) nothing

Dealing with NULL

COALESCE(a, b, ...)

Return the first non NULL value of the list (or NULL).

Dealing with NULL

COALESCE(a, b, ...)

Return the first non NULL value of the list (or NULL).

---

v **ISNULL**

Return a boolean determining whether *v* is NULL

Dealing with NULL

COALESCE(a, b, ...)

Return the first non NULL value of the list (or NULL).

---

v **ISNULL**

Return a boolean determining whether *v* is NULL

The logic in SQL is three-valued True, False, and NULL.

WHERE ... IN

Useful to test values within a set of values

```
SELECT * FROM table
WHERE someCol IN (1,23,565,3)
```

WHERE EXISTS / WHERE NOT EXISTS

Useful to test conditions over tables

```
SELECT * FROM table t1
WHERE NOT EXISTS (
      SELECT *
      FROM otherTable t2
      WHERE t2.someCol == t1.otherCol
)
```

Exercises

A - Average score for each movie

B - Ids of the movies with an average over 4

C - List of Ids of movies ordered by average score

D - Ids of movies with a rating but no title

E - Titles of the 10 best movies

F - Titles of the 10 to 20 best movies (20 best ones minus the 10 best)

G - Titles of the 10 best movies according to the score:

$$\frac{\sum votes}{\text{nb}(votes) + 1}$$

Evaluation and optimization of SQL queries

Optimization pipeline

Query is translated into a logical representation

Optimization pipeline

Query is translated into a logical representation

$$\Downarrow$$

Optimization pipeline

Query is translated into a logical representation

$\Downarrow$

We find alternative representations for the query

Optimization pipeline

Query is translated into a logical representation

$\Downarrow$

We find alternative representations for the query

$\Downarrow$

Optimization pipeline

Query is translated into a logical representation

$\Downarrow$

We find alternative representations for the query

$\Downarrow$

A cost estimator finds the best way to execute the query

Optimization pipeline

Query is translated into a logical representation

$\Downarrow$

We find alternative representations for the query

$\Downarrow$

A cost estimator finds the best way to execute the query

$\Downarrow$

Optimization pipeline

Query is translated into a logical representation

$\Downarrow$

We find alternative representations for the query

$\Downarrow$

A cost estimator finds the best way to execute the query

$\Downarrow$

The query is executed

Evaluation of SQL queries 1/2

SELECT * FROM movies WHERE userId = 0;

```
pguser=> EXPLAIN SELECT * FROM ratings WHERE userId = 0  ;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on ratings  (cost=0.00..1903.45 rows=79 width=24)
   Filter: (userid = 0)
(2 rows)
```

Evaluation of SQL queries 2/2

SELECT * FROM movies WHERE title LIKE 'Jumanji%';

```
pguser=> EXPLAIN
      SELECT * FROM movies
      WHERE title LIKE 'Jumanji%' ;
                    QUERY PLAN
-------------------------------------------------------
 Seq Scan on movies  (cost=0.00..218.76 rows=1 width=48)
  Filter: (title ~~ 'Juman%'::text)
    (2 rows)
```

Index

Optimization of SQL queries

One of the great advantage of using SQL is to let the query engine optimize the queries.

Optimization of SQL queries

One of the great advantage of using SQL is to let the query engine optimize the queries.

To really optimize the queries, the engine needs indexes!

Different types of indexes

Default: btree

Retrieves efficiently by value or order

Different types of indexes

Default: btree

Retrieves efficiently by value or order

Hash

Retrieves efficiently by value

Different types of indexes

Default: btree

Retrieves efficiently by value or order

Hash

Retrieves efficiently by value

GiST / SP-GiST

Retrieves efficiently geographical data

Different types of indexes

Default: btree

Retrieves efficiently by value or order

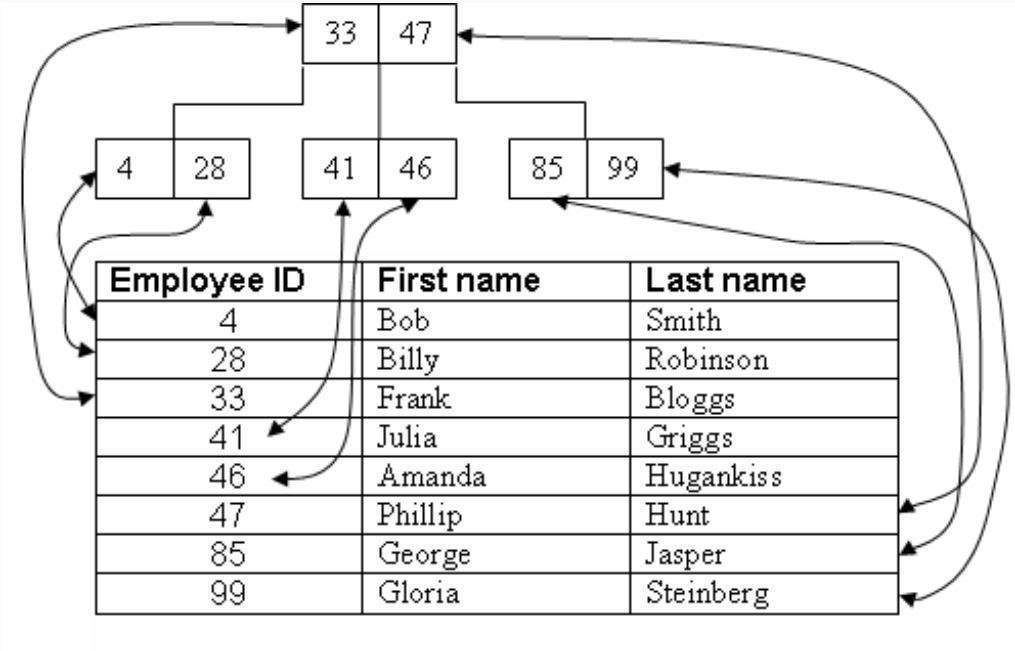
Hash

Retrieves efficiently by value

GiST / SP-GiST

Retrieves efficiently geographical data

Example of index



Source `https://en.wikipedia.org/wiki/Architecture_of_Btrieve`

Indexes on ratings

```
                                    Table "public.ratings"
 Column  |       Type       | Collation | Nullable | Default | Storage | Stats target | Description
---------+------------------+-----------+----------+---------+---------+--------------+-------------
 userid  | integer          |           |          |         | plain   |              |
 movieid | integer          |           |          |         | plain   |              |
 rating  | double precision |           |          |         | plain   |              |
 time    | bigint           |           |          |         | plain   |              |
Indexes:
    "ratings_movieid_idx" btree (movieid)

                                  Table "public.movies"
 Column |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+---------+-----------+----------+---------+----------+--------------+-------------
 id     | integer |           |          |         | plain    |              |
 title  | text    |           |          |         | extended |              |
 genre  | text    |           |          |         | extended |              |
Indexes:
    "titleIdx" btree (title)
    "titleIdxTxt" btree (title text_pattern_ops)
    "movies_id_idx" btree (id)
```

SELECT * FROM movies WHERE movieId = 0;

```
pguser=> EXPLAIN SELECT * FROM ratings WHERE movieId = 0  ;
                                    QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on ratings  (cost=4.39..51.01 rows=13 width=24)
   Recheck Cond: (movieid = 0)
   ->  Bitmap Index Scan on ratings_movieid_idx
                                   (cost=0.00..4.39 rows=13 width=0)
         Index Cond: (movieid = 0)
(4 rows)
```

Evaluation of SQL queries

SELECT * FROM movies WHERE title LIKE 'Jumanji%';

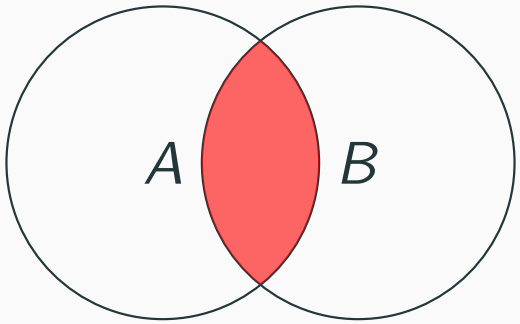
```
pguser=> EXPLAIN SELECT * FROM movies WHERE title LIKE 'Jumanji%' ;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using titleIdxTxt on movies  (cost=0.29..8.31 rows=1 width=48)
   Index Cond: ((title ~>=~ 'Jumanji'::text) AND (title ~<~ 'Jumanjj'::text))
   Filter: (title ~~ 'Jumanji%'::text)
(3 rows)
```

Join

INNER JOIN

```sql
SELECT *
FROM myTable, mySecondTable
WHERE myTable.someCol = mySecondTable.someCol
```

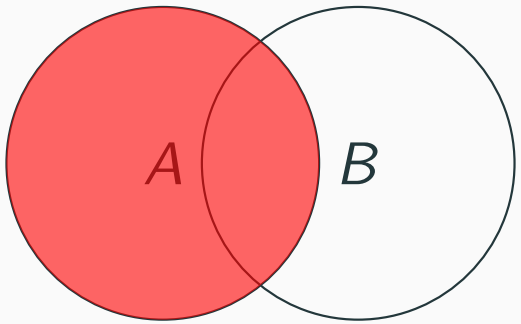
```sql
SELECT *
FROM myTable
INNER JOIN  mySecondTable
ON myTable.someCol = mySecondTable.someCol
```



LEFT JOIN

```
SELECT *
FROM myTable
LEFT JOIN  mySecondTable
ON myTable.someCol = mySecondTable.someCol
```

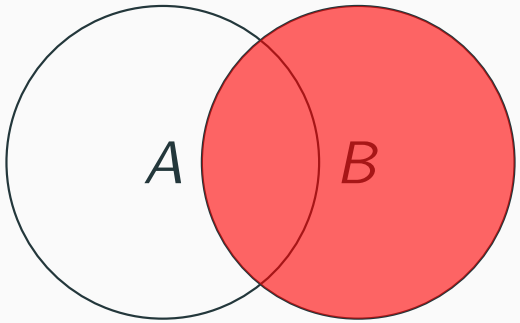
Includes the INNER JOIN + all elements from myTable with no match in mySecondTable.



RIGHT JOIN

```
SELECT *
FROM myTable
LEFT JOIN  mySecondTable
ON myTable.someCol = mySecondTable.someCol
```

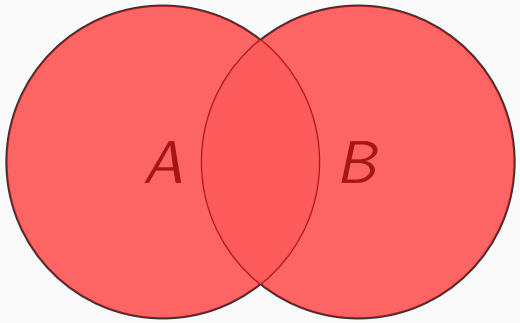
Includes the INNER JOIN + all elements from myTable with no match in mySecondTable.



FULL JOIN

```
SELECT *
FROM myTable
FULL JOIN  mySecondTable
ON myTable.someCol = mySecondTable.someCol
```

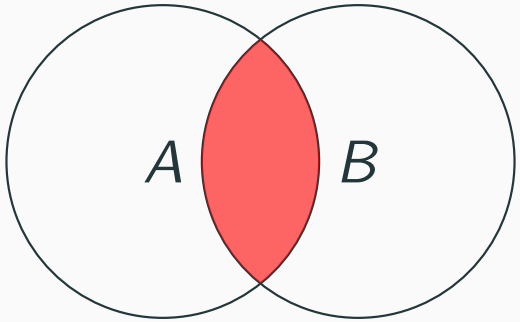
Includes the INNER JOIN + all elements with no match.



NATURAL JOIN

```
SELECT *
FROM myTable
NATURAL JOIN  mySecondTable
```

The INNER JOIN with condition on default columns.



UNION / UNION ALL

```
SELECT *
FROM myTable
UNION
SELECT *
FROM mySecondTable
```

UNION in *the set sense*!

UNION / UNION ALL

```
SELECT *
FROM myTable
UNION
SELECT *
FROM mySecondTable
```

UNION in *the set sense*!

```
SELECT *
FROM myTable
UNION ALL
SELECT *
FROM mySecondTable
```

UNION in *the multiset sense*!

MINUS / EXPECT

```
SELECT *
FROM myTable
EXCEPT
SELECT *
FROM mySecondTable
```

Difference