

Introduction to Relational Database Management Systems

Pierre Senellart



4 September 2020
PSL Preparatory Week

Outline

Introduction

Data management

Relational Databases

SQL

Other data models

Conclusion

Data management

Numerous applications (standalone software, Web sites, etc.) need to **manage data**:

- **Structure** data useful to the application
- Store them in a **persistent** manner (data retained even when the application is not running)
- **Efficiently query** information within large data volumes
- **Update** data without violating some structural **constraints**
- Enable data access and updates by **multiple users**, possibly **concurrently**

Often, desirable to access the same data from **several distinct applications**, from distinct computers.

Example: Hotel information system

Access from custom software (front desk), a Web site (customers), some accounting software. Requirements:

- **Structured data** representing rooms, customers, reservations, rates, etc.
- **No data lost** when these applications are not used, or when a general power cut arises
- **Find quasi-instantaneously** which rooms are booked, by whom, on a given date, in a history of several years of reservations
- Easily **add** a reservation while making sure the same room is not booked twice the same day
- The customer, the front desk agent, the accountant, must not have the same **view** of data (confidentiality, ease of use, etc.); different customers cannot book the same room **at the same instant**

Naïve implementation (1/2)

- Implementing in a classical programming language (C++, Java, Python, etc.) data structures to represent all useful data
- Defining ad-hoc file formats to store data on disk, with regular synchronization and a mechanism to retrieve data in case of failure
- Storing data in the memory of the application, with data structures (binary search trees, hash tables) and algorithms (search, sorting, aggregation, graph traversal, etc.) allowing to find data efficiently
- Update functionalities, with code checking on the fly that no business constraint is violated

Naïve implementation (2/2)

- Defining within the software different user roles, an authentication mechanism; using threads to answer different queries at the same time, locks/semaphores on data manipulation functions with possible race conditions
- Defining and implementing a communication protocol to connect this software component to a Web server, some desktop software, a business accounting suite, etc.

Naïve implementation (2/2)

- Defining within the software different user roles, an authentication mechanism; using threads to answer different queries at the same time, locks/semaphores on data manipulation functions with possible race conditions
- Defining and implementing a communication protocol to connect this software component to a Web server, some desktop software, a business accounting suite, etc.

Lots of work!

Naïve implementation (2/2)

- Defining within the software different user roles, an authentication mechanism; using threads to answer different queries at the same time, locks/semaphores on data manipulation functions with possible race conditions
- Defining and implementing a communication protocol to connect this software component to a Web server, some desktop software, a business accounting suite, etc.

Lots of work! Needs a programmer that masters OOP, serialization, failover, data structures, algorithms, integrity constraints, role management, parallel programming, concurrency control, networking. . .

Naïve implementation (2/2)

- Defining within the software different user roles, an authentication mechanism; using threads to answer different queries at the same time, locks/semaphores on data manipulation functions with possible race conditions
- Defining and implementing a communication protocol to connect this software component to a Web server, some desktop software, a business accounting suite, etc.

Lots of work! Needs a programmer that masters OOP, serialization, failover, data structures, algorithms, integrity constraints, role management, parallel programming, concurrency control, networking... and this must be redone **for every single application** that manages data!

Role of a DBMS

Database Management System

Software that **simplifies the design** of applications that handle data, by providing a **unified access** to the functionalities required for **data management**, whatever the application.

Database

Collection of data (specific to a given application) managed by a DBMS

Major types of DBMSs

Relational (RDBMS). Tables, complex queries (SQL), rich features

XML. Trees, complex queries (XQuery), features similar to RDBMS

Graph/Triples. Graph data, complex queries expressing graph navigation

Objects. Complex data model, inspired by OOP

Documents. Complex data, organized in documents, relatively simple queries and features

Key-Value. Very basic data model, focus on performance

Column Stores. Data model in between key-value and RDBMS; focus on iteration and aggregation on columns

Major types of DBMSs

Relational (RDBMS). Tables, complex queries (SQL), rich features

XML. Trees, complex queries (XQuery), features similar to RDBMS

Graph/Triples. Graph data, complex queries expressing graph navigation

Objects. Complex data model, inspired by OOP

Documents. Complex data, organized in documents, relatively simple queries and features

Key-Value. Very basic data model, focus on performance

Column Stores. Data model in between key-value and RDBMS; focus on iteration and aggregation on columns

NoSQL

Outline

Introduction

Relational Databases

The relational model

The relational algebra

SQL

Other data models

Conclusion

Classical relational DBMSs

- Based on the **relational model**: decomposition of data into relations (i.e., tables)
- A standard query language: **SQL**
- Data **stored on disk**
- Relations (tables) stored **line after line**
- **Centralized** system, with limited distribution possibilities

ORACLE®



PostgreSQL



Relational schema

We fix countably infinite sets:

- \mathcal{L} of labels
- \mathcal{V} of values
- \mathcal{T} of types, s.t., $\forall \tau \in \mathcal{T}, \tau \subseteq \mathcal{V}$

Definition

A **relation schema** (of **arity** n) is an n -tuple (A_1, \dots, A_n) where each A_i (called an **attribute**) is a pair (L_i, τ_i) with $L_i \in \mathcal{L}$, $\tau_i \in \mathcal{T}$ and such that all L_i are distinct

Definition

A **database schema** is defined by a finite set of labels $L \subseteq \mathcal{L}$ (**relation names**), each label of L being mapped to a relation schema.

Example database schema

- Universe:
 - \mathcal{L} the set of alphanumeric character strings starting with a letter
 - \mathcal{V} the set of finite sequences of bits
 - \mathcal{T} is formed of types such as INTEGER (representation as a sequence of bits of integers between -2^{31} and $2^{31} - 1$), REAL (representation of floating-point numbers following IEEE 754), TEXT (UTF-8 representation of character strings), DATE (ISO8601 representation of dates), etc.
- Database schema formed of 2 relation names, Guest and Reservation
- Guest: ((id, INTEGER), (name, TEXT), (email, TEXT))
- Reservation:
((id, INTEGER), (guest, INTEGER), (room, INTEGER),
(arrival, DATE), (nights, INTEGER))

Database

Definition

An **instance** of a relation schema $((L_1, \tau_1), \dots, (L_n, \tau_n))$ (also called a **relation on this schema**) is a **finite set** $\{t_1, \dots, t_k\}$ of tuples of the form $t_j = (v_{j1}, \dots, v_{jn})$ with $\forall j \forall i v_{ji} \in \tau_i$.

Definition

An **instance** of a database schema (or, simply, a **database on this schema**) is a function that maps each relation name to an instance of the corresponding relation schema.

Note: **Relation** is used somewhat ambiguously to talk about a relation schema or an instance of a relation schema.

Example

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Some notation

- If $A = (L, \tau)$ is the i th attribute of a relation R , and t an n -tuple of an instance of R , we denote $t[A]$ (or $t[L]$) the value of the i th component of t .
- Similarly, if \mathcal{A} is a k -tuple of attributes appearing within the n attributes of R , $t[\mathcal{A}]$ is the k -tuple formed from t by concatenating the $t[A]$'s for A in \mathcal{A} .

Simple integrity constraints

We can add to the relational schema some **integrity constraints**, of different types, to define a notion of instance **validity**.

Key. A tuple of attributes \mathcal{A} of a relation schema R is a **key** if there cannot be two distinct tuples t_1 and t_2 in an instance of R with $t_1[\mathcal{A}] = t_2[\mathcal{A}]$

Foreign key. A k -tuple of attributes \mathcal{A} of a relation schema R is a **foreign key referencing** a k -tuple of attributes \mathcal{B} of a relation S if for all instances I^R and I^S of R and S , for every tuple t of I^R , there exists a **unique** tuple t' of I^S with $t[\mathcal{A}] = t'[\mathcal{B}]$

Check constraint. Arbitrary condition on the values of the attributes of a relation (applying to every tuple of the instances of that relation)

Examples of constraints

- id is a **key** of Guest
- email is a **key** of Guest
- id is a **key** of Reservation
- (room, arrival) is a **key** of Reservation
- (guest, arrival) is a **key** of Reservation (?)
- guest is a **foreign key** of Reservation referencing id of Guest
- In Guest, email **must** contain an “@”
- In Reservation, room **must** be between 1 and 650
- In Reservation, nights **must** be positive

Impossible to express more complex constraints (e.g., a room can only be occupied once the same night, which would require comparing the arrival date and number of nights of different tuples with the same room)

Variant: bag semantics

- A relation instance is defined as a (finite) set of tuples. One can also consider a **bag semantics** of the relational model, where a relation instance is a multiset of tuples.
- This is what best matches how RDBMSs work...
- ... but most of relational database theory has been established for the set semantics, **more convenient** to work with
- We will **mostly discuss the set semantics** in this lecture

Outline

Introduction

Relational Databases

The relational model

The relational algebra

SQL

Other data models

Conclusion

The relational algebra

- **Algebraic language** to express queries
- A relational algebra expression produces a **new relation** from the database relations
- Each operator takes 0, 1, or 2 **subexpressions**
- Main operators:

Op.	Arity	Description	Condition
R	0	Relation name	$R \in \mathcal{L}$
$\rho_{A \rightarrow B}$	1	Renaming	$A, B \in \mathcal{L}$
$\Pi_{A_1 \dots A_n}$	1	Projection	$A_1 \dots A_n \in \mathcal{L}$
σ_φ	1	Selection	φ formula
\times	2	Cross product	
\cup	2	Union	
\setminus	2	Difference	
\bowtie_φ	2	Join	φ formula

Relation name

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: Guest

Result:

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Renaming

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: $\rho_{id \rightarrow guest}(\text{Guest})$

Result:

guest	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Projection

Guest		
id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation				
id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\text{email}, \text{id}}(\text{Guest})$

Result:

email	id
john.smith@gmail.com	1
alice@black.name	2
john.smith@ens.fr	3

Selection

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: $\sigma_{\text{arrival} > 2017-01-12 \wedge \text{guest} = 2}(\text{Reservation})$

Result:

id	guest	room	arrival	nights
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

The formula used in the selection can be any **Boolean combination** of **comparisons** of attributes to attributes or constants.

Cross product

Guest		
id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation				
id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{id}(\text{Guest}) \times \Pi_{name}(\text{Guest})$

Result:

id	name
1	Alice Black
2	Alice Black
3	Alice Black
1	John Smith
2	John Smith
3	John Smith

Union

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: $\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation})) \cup$
 $\Pi_{\text{room}}(\sigma_{\text{arrival}=2017-01-15}(\text{Reservation}))$

Result:

room
107
302
504

Union

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: $\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation})) \cup$
 $\Pi_{\text{room}}(\sigma_{\text{arrival}=2017-01-15}(\text{Reservation}))$

Result:

room
107
302
504

This simple union could have been written

$\Pi_{\text{room}}(\sigma_{\text{guest}=2 \vee \text{arrival}=2017-01-15}(\text{Reservation}))$. Not always possible.

Difference

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: $\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation})) \setminus$
 $\Pi_{\text{room}}(\sigma_{\text{arrival}=2017-01-15}(\text{Reservation}))$

Result:
 room

 107

Difference

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: $\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation})) \setminus$
 $\Pi_{\text{room}}(\sigma_{\text{arrival}=2017-01-15}(\text{Reservation}))$

Result:
 room
 107

This simple difference could have been written $\Pi_{\text{room}}(\sigma_{\text{guest}=2 \wedge \text{arrival} \neq 2017-01-15}(\text{Reservation}))$. Not always possible.

Join

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Expression: `Reservation ⋈guest=id Guest`

Result:

id	guest	room	arrival	nights	name	email
1	1	504	2017-01-01	5	John Smith	john.smith@gmail.com
2	2	107	2017-01-10	3	Alice Black	alice@black.name
3	3	302	2017-01-15	6	John Smith	john.smith@ens.fr
4	2	504	2017-01-15	2	Alice Black	alice@black.name
5	2	107	2017-01-30	1	Alice Black	alice@black.name

The formula used in the join can be any **Boolean combination** of **comparisons** of attributes of the table on the left to attributes of the table on the right.

Note on the join

- The join is not an **elementary** operator of the relational algebra (but it is very useful)
- It can be seen as a **combination** of renaming, cross product, selection, projection
- Thus:

$$\begin{aligned} & \text{Reservation} \bowtie_{\text{guest=id}} \text{Guest} \\ \equiv & \Pi_{\text{id,guest,room,arrival,nights,name,email}}(\\ & \sigma_{\text{guest=temp}}(\text{Reservation} \times \rho_{\text{id} \rightarrow \text{temp}}(\text{Guest}))) \end{aligned}$$

- If R and S have for attributes \mathcal{A} and \mathcal{B} , we note $R \bowtie S$ the **natural join** of R and S , where the join formula is

$$\bigwedge_{A \in \mathcal{A} \cap \mathcal{B}} A = A.$$

Illegal operations

- All expressions of the relational algebra are not **valid**
- The validity of an expression generally depends on the relational schema
- For example:
 - One cannot refer to a relation name that does not exist in the relational schema
 - One cannot refer (within renaming, projection, selection, join) to an attribute that does not exist in the result of a sub-expression
 - One cannot union two relations with different attributes
 - One cannot build (cross product, join, renaming) a table with two attributes with the same name
- Systems implementing the relational algebra can perform static or dynamic checks of these rules, or sometimes ignore them

Bag semantics

In bag semantics (what is actually used by RDBMS):

- All operations return **multisets**
- In particular, projection and union can **introduce** multisets even when initial relations are sets

Extension: Aggregation

- Various extensions have been proposed to the relational algebra to add **additional features**
- In particular, **aggregation and grouping** [Klug, 1982, Libkin, 2003] of results
- With a syntax inspired from [Libkin, 2003]:

$$\sigma_{\text{avg} > 3}(\gamma_{\text{room}}^{\text{avg}}[\lambda x. \text{avg}(x)](\Pi_{\text{room}, \text{nights}}(\text{Reservation})))$$

computes the average number of nights per reservation for each room having an average greater than 3

room	avg
302	6
504	3.5

Outline

Introduction

Relational Databases

SQL

Basics and DDL

DML

Other data models

Conclusion

SQL

- **Structured Query Language**, standardized language (ISO/IEC 9075, several versions [ISO, 1987, 1999]) to **interact with an RDBMS**
- Unfortunately, implementation of the standard very **variable** from one RDBMS to the other
- Many little things (e.g., available types) vary between DBMSs instead of following the standard
- Differences are more **syntactical** than essential
- Where there is a difference, we give the **PostgreSQL** version
- Two main parts: DDL (**Data Definition Language**) to define the schema and DML (**Data Manipulation Language**) to query and update the database
- **Declarative** language: one writes what one wants, the system is free to transform what was written into an **efficient execution plan**

SQL syntax

- Quite **verbose**, designed to be almost English-like [Chamberlin and Boyce, 1974]
- Keywords **case-insensitive**, traditionally written in uppercase
- Identifiers often **case-insensitive** (depends on the RDBMS)
- **Comments** introduced by --
- SQL statements terminated by a “;” in certain contexts but the “;” is not strictly part of the SQL statement

NULL

- In SQL, NULL is a special value that any attributes of a tuple can take
- Denotes the **absence of a value**
- Different from 0, the empty string, etc.
- Weird **tri-valued** logic: True, False, NULL
- A regular comparison (equality, inequality...) with NULL always returns NULL
- **IS NULL, IS NOT NULL** can be used to test if a value is NULL
- NULL is eventually converted into False
- Weird consequences, poor integration with the formal relational model

Data Definition Language

```
CREATE TABLE Guest(id INTEGER, name TEXT, email TEXT);  
CREATE TABLE Reservation(id INTEGER, guest INTEGER,  
    room INTEGER, arrival DATE, nights INTEGER);
```

But also:

- **DROP TABLE** Guest; to delete a table
- **ALTER TABLE** Guest **RENAME TO** guest2; to rename a table
- **ALTER TABLE** Guest **ALTER COLUMN** id **TYPE** TEXT; to change the type of a column

Constraints (1/2)

Specified when the table is created, or added afterwards (with **ALTER TABLE**)

PRIMARY KEY for the primary key; only one per table (but can include several attributes), it's a key that will be used for physical organization of data; implies **NOT NULL**

UNIQUE for other keys

REFERENCES for foreign keys

CHECK for Check constraints

NOT NULL to indicate that an attribute cannot be NULL

Constraints (2/2)

```
CREATE TABLE Guest(  
  id INTEGER PRIMARY KEY,  
  name TEXT NOT NULL,  
  email TEXT UNIQUE CHECK (email LIKE '%@%')  
);
```

```
CREATE TABLE Reservation(  
  id INTEGER PRIMARY KEY,  
  guest INTEGER NOT NULL REFERENCES Guest(id),  
  room INTEGER NOT NULL CHECK (room>0  
    AND room<651),  
  arrival DATE NOT NULL,  
  nights INTEGER NOT NULL CHECK (nights>0),  
  UNIQUE(room, arrival),  
  UNIQUE(guest, arrival));
```

Outline

Introduction

Relational Databases

SQL

Basics and DDL

DML

Other data models

Conclusion

Updates

- Insertions:

```
INSERT INTO Guest(id,name) VALUES (5, 'John');
```

- Deletions:

```
DELETE FROM Reservation WHERE id>4;
```

- Modifications:

```
UPDATE Reservation  
  SET room=205  
  WHERE room=204;
```

Inserting several values

INSERT INTO Guest VALUES

```
(1, 'Jean Dupont', 'jean.dupont@gmail.com'),  
(2, 'Alice Dupuis', 'alice@dupuis.name'),  
(3, 'Jean Dupont', 'jean.dupont@ens.fr');
```

INSERT INTO Reservation VALUES

```
(1, 1, 504, '2017-01-01', 5),  
(2, 2, 107, '2017-01-10', 3),  
(3, 3, 302, '2017-01-15', 6),  
(4, 2, 504, '2017-01-15', 2),  
(5, 2, 107, '2017-01-30', 1);
```


Queries

Following general form:

SELECT ... **FROM** ... **WHERE** ...
GROUP BY ... **HAVING** ...
UNION SELECT ... **FROM** ...

SELECT projection, renaming, aggregation

FROM cross product, join

WHERE selection, join (optional)

GROUP BY grouping (optional)

HAVING selection on the (aggregated) result of the
grouping (optional)

UNION union (optional)

Other keywords: **ORDER BY** to reorder, **LIMIT** to limit to the
 k first results, **DISTINCT** to force set semantics, **EXCEPT** for
difference...

Renaming

Guest		
id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation				
id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

$$\rho_{id \rightarrow guest}(\text{Guest})$$

```
SELECT id AS guest, name, email
FROM Guest;
```

Projection

Guest		
id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation				
id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

$$\Pi_{\text{email, id}}(\text{Guest})$$

```
SELECT DISTINCT email, id  
FROM Guest;
```

Selection

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

$\sigma_{\text{arrival} > 2017-01-12 \wedge \text{guest}=2}(\text{Reservation})$

```
SELECT *  
FROM Reservation  
WHERE arrival > '2017-01-12' AND guest=2;
```

Cross product

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

$$\Pi_{id}(\text{Guest}) \times \Pi_{name}(\text{guest})$$

```
SELECT *  
FROM  
  (SELECT DISTINCT id FROM Guest) AS temp1,  
  (SELECT DISTINCT name FROM Guest) AS temp2  
ORDER BY name, id;
```

Union

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

$$\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation}))$$
$$\cup \Pi_{\text{room}}(\sigma_{\text{arrival}=2017-01-15}(\text{Reservation}))$$

```
SELECT room
FROM Reservation
WHERE guest=2
UNION
SELECT room
FROM Reservation
WHERE arrival='2017-01-15';
```

Difference

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

$$\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation}))$$
$$\setminus \Pi_{\text{room}}(\sigma_{\text{arrival}=2017-01-15}(\text{Reservation}))$$

```
SELECT room
FROM Reservation
WHERE guest=2
EXCEPT
SELECT room
FROM Reservation
WHERE arrival='2017-01-15';
```

Join

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

Reservation ⋈_{guest=id} Guest

```
SELECT Reservation.*, name, email  
FROM Reservation JOIN Guest ON guest=Client.id;
```

```
SELECT Reservation.*, name, email  
FROM Reservation, Guest  
WHERE guest=Guest.id;
```


Aggregation

Guest			Reservation				
id	name	email	id	guest	room	arrival	nights
1	John Smith	john.smith@gmail.com	1	1	504	2017-01-01	5
2	Alice Black	alice@black.name	2	2	107	2017-01-10	3
3	John Smith	john.smith@ens.fr	3	3	302	2017-01-15	6
			4	2	504	2017-01-15	2
			5	2	107	2017-01-30	1

$$\sigma_{\text{avg} > 3}(\gamma_{\text{room}}^{\text{avg}}[\lambda x. \text{avg}(x)](\Pi_{\text{room}, \text{nights}}(\text{Reservation})))$$

```
SELECT room, AVG(nights) AS avg
FROM Reservation
GROUP BY room
HAVING AVG(nights) > 3
ORDER BY room;
```

Outline

Introduction

Relational Databases

SQL

Other data models

Limitations of classical RDMSs

NoSQL and NewSQL

Conclusion

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution
- **Complex** queries

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution
- **Complex** queries
- Very fine query **optimization**, **index** allowing quick access to data

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution
- **Complex** queries
- Very fine query **optimization**, **index** allowing quick access to data
- **Mature**, **stable**, **efficient** software, wealth of features and of interfaces

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution
- **Complex** queries
- Very fine query **optimization**, **index** allowing quick access to data
- **Mature**, **stable**, **efficient** software, wealth of features and of interfaces
- **Integrity constraints** ensuring invariants on data

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution
- **Complex** queries
- Very fine query **optimization**, **index** allowing quick access to data
- **Mature**, **stable**, **efficient** software, wealth of features and of interfaces
- **Integrity constraints** ensuring invariants on data
- Efficient management of **large data volume** (gigabytes, even terabytes)

Strengths of classical relational DBMSs

- **Independence** between:
 - data model and storage structure
 - declarative queries and execution
- **Complex** queries
- Very fine query **optimization**, **index** allowing quick access to data
- **Mature**, **stable**, **efficient** software, wealth of features and of interfaces
- **Integrity constraints** ensuring invariants on data
- Efficient management of **large data volume** (gigabytes, even terabytes)
- **Transactions** (set of elementary operations) for concurrency control, user isolation, error recovery

ACID Properties

Transactions of classical relational DBMSs respect the **ACID** properties:

ACID Properties

Transactions of classical relational DBMSs respect the **ACID** properties:

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

ACID Properties

Transactions of classical relational DBMSs respect the **ACID** properties:

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Consistency: Transactions respect integrity constraints of the database

ACID Properties

Transactions of classical relational DBMSs respect the **ACID** properties:

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Consistency: Transactions respect integrity constraints of the database

Isolation: Two concurrent executions of transactions result in a state equivalent to serial execution

ACID Properties

Transactions of classical relational DBMSs respect the **ACID** properties:

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Consistency: Transactions respect integrity constraints of the database

Isolation: Two concurrent executions of transactions result in a state equivalent to serial execution

Durability: Once a transaction is committed, data remain durably stored in the database, even in case of (e.g., hardware) failure

Weaknesses of classical relational DBMSs

- Inability to manage **very large data volumes** (order of magnitude of petabytes)

Weaknesses of classical relational DBMSs

- Inability to manage **very large data volumes** (order of magnitude of petabytes)
- Impossible to manage **extreme loads** (thousands of queries per seconds and more)

Weaknesses of classical relational DBMSs

- Inability to manage **very large data volumes** (order of magnitude of petabytes)
- Impossible to manage **extreme loads** (thousands of queries per seconds and more)
- The relational model is not suitable to storage and querying of **some data types** (hierarchical data, weakly structured data, semi-structured data)

Weaknesses of classical relational DBMSs

- Inability to manage **very large data volumes** (order of magnitude of petabytes)
- Impossible to manage **extreme loads** (thousands of queries per seconds and more)
- The relational model is not suitable to storage and querying of **some data types** (hierarchical data, weakly structured data, semi-structured data)
- ACID properties lead to serious **overheads** in latency, disk access, CPU time (locks, logging, etc.)

Weaknesses of classical relational DBMSs

- Inability to manage **very large data volumes** (order of magnitude of petabytes)
- Impossible to manage **extreme loads** (thousands of queries per seconds and more)
- The relational model is not suitable to storage and querying of **some data types** (hierarchical data, weakly structured data, semi-structured data)
- ACID properties lead to serious **overheads** in latency, disk access, CPU time (locks, logging, etc.)
- Performance **limited by disk accesses**

Outline

Introduction

Relational Databases

SQL

Other data models

Limitations of classical RDMSs

NoSQL and NewSQL

Conclusion

NoSQL

- **No SQL** or **Not Only SQL**
- DBMSs with other trade-offs than those made by classical systems
- **Very diversified** ecosystem
- **Desiderata**: different data model, transparent scaling up, extreme performances
- **Features abandoned**: strong concurrency control and consistency, (possibly) complex queries



Systems with a different data model

Complex queries, non-relational data model

Type	Organization	Queries	Examples of systems
------	--------------	---------	---------------------




Systems with a different data model

Complex queries, non-relational data model

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	 





Systems with a different data model

Complex queries, non-relational data model

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	 
Object	Complex data, with properties and methods	OQL, VQL	 VERSANT







Systems with a different data model

Complex queries, non-relational data model

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	 
Object	Complex data, with properties and methods	OQL, VQL	 VERSANT
Graph	Graph with vertices, edges, labels	Cypher, Gremlin	 Neo4j the graph database

Systems with a different data model

Complex queries, non-relational data model

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	 
Object	Complex data, with properties and methods	OQL, VQL	 VERSANT
Graph	Graph with vertices, edges, labels	Cypher, Gremlin	 Neo4j the graph database
Triples	RDF triples from the Semantic Web	SPARQL	 

Key-value stores

- **Very simple** queries:
 - `get` retrieves the value mapped to a key
 - `put` adds a new key/value pair
- Stress put on transparent **scaling up**, **low latency**, **very high bandwidth**
- Example of implementation: **distributed hash table**

Amazon DynamoDB



Chord

MemcacheDB

Document stores

- Still **very simple** queries:
 - `get` retrieves the document (JSON, XML, YAML) mapped to a key
 - `put` maps a new document to a key
- **Additional indexes** allow retrieval of documents containing a keyword, having a given property, etc.
- Documents **organized in collections**, metadata (versions, dates) management, etc.
- Accent put on **interface simplicity**, **ease of handling** in a programming language



Column stores

- Instead of storing data row after row, store it **column after column**
- **Richer** organization than key-value stores (several column by stored object)
- Makes **aggregating or scanning the values of a given column** more efficient
- Transparent **distribution, scaling up** thanks to distributed search trees or distributed hash tables



NewSQL

- Some applications require:
 - **rich** query languages (joins, aggregation)
 - conformity to **ACID** properties
 - but **higher performances** than classical DBMSs

NewSQL

- Some applications require:
 - rich query languages (joins, aggregation)
 - conformity to ACID properties
 - but higher performances than classical DBMSs
- Possible solutions:
 - Get rid of classical bottlenecks of DBMSs: locks, logging, cache management
 - Main-memory database, with asynchronous copy to disk
 - Lock-free concurrence management (MVCC)
 - Shared-nothing distributed architecture, transparent load balancing



When to choose a non-classical DBMS?

- Extreme latency or bandwidth requirements

When to choose a non-classical DBMS?

- Extreme latency or bandwidth requirements
- Extreme data volumes

When to choose a non-classical DBMS?

- Extreme latency or bandwidth requirements
- Extreme data volumes
- When the relational model and SQL poorly suit storage and data access needs (not that frequent!)

When to choose a non-classical DBMS?

- Extreme latency or bandwidth requirements
- Extreme data volumes
- When the relational model and SQL poorly suit storage and data access needs (not that frequent!)
- When, after extensive tests, performances of classical DBMSs prove insufficient

When to choose a non-classical DBMS?

- **Extreme latency** or **bandwidth** requirements
- **Extreme data volumes**
- When the relational model and SQL **poorly suit** storage and data access needs (not that frequent!)
- When, after extensive tests, performances of classical DBMSs prove **insufficient**
- **Know what you lose**: (depending on the case) ACID, possibility of complex querying, stability of well-established software, etc.

When to choose a non-classical DBMS?

- **Extreme latency** or **bandwidth** requirements
- **Extreme data volumes**
- When the relational model and SQL **poorly suit** storage and data access needs (not that frequent!)
- When, after extensive tests, performances of classical DBMSs prove **insufficient**
- **Know what you lose**: (depending on the case) ACID, possibility of complex querying, stability of well-established software, etc.
- NoSQL and NewSQL databases answer **real needs** but needs are **often overestimated**

Outline

Introduction

Relational Databases

SQL

Other data models

Conclusion

References

References

- **Generalities** on data management [Benedikt and Senellart, 2012, Abiteboul, 2012]
- Course (in French) on the curriculum in databases of the “classes préparatoires” [Abiteboul et al., 2014]
- Relational **model**, relational **algebra**: Chapters 3 and 4 of [Abiteboul et al., 1995]
- **Details of SQL**: standards are not public and not very informative for the final user; use the documentation of the DBMS
- For **PostgreSQL**, <https://www.postgresql.org/docs/> and \help in the command-line client

Bibliography I

Serge Abiteboul. *Sciences des données: de la logique du premier ordre à la Toile*. Collège de France, 2012.
<http://books.openedition.org/cdf/529>.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

Serge Abiteboul, Benjamin Nguyen, and Yannick Le Bras. Introduction aux bases de données relationnelles.
<http://abiteboul.com/Lili/bdrelationnelles.pdf>, 2014.

Michael Benedikt and Pierre Senellart. Databases. In Edward K. Blum and Alfred V. Aho, editors, *Computer Science. The Hardware, Software and Heart of It*, pages 169–229. Springer-Verlag, 2012.

Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proc. SIGFIDET/SIGMOD Workshop*, volume 1, 1974.

Bibliography II

ISO. *ISO 9075:1987: SQL*. International Standards Organization, 1987.

ISO. *ISO 9075:1999: SQL*. International Standards Organization, 1999.

Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003.