

# Data acquisition, extraction, and storage

## Handling Relational Data

Pierre Senellart



17 October 2024



# Plan

## Introduction

### Data Management

#### Types of DBMSs

## Introduction to The Relational Model

## Recursive Queries

## DB or not DB?

## References

## 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: Information system of a hotel

Access from an in-house software (front desk), a Website (guests), an accounting software suite. Requirements:

- **Structured data** representing rooms, customers, guests, bookings, rates, etc.
- **No loss of data** when these applications are unused, or when a power cut arises
- **Find quasi-instantaneously** which rooms are booked, by whom, a given day, within a history containing several years of bookings
- Easily **add** a booking by ensuring the same room is not booked twice the same day
- The guest, the front desk employee, the accountant, must not have the same **view** of the data (confidentiality, ease of use, etc.)
- If a room is available, it cannot be booked by different guests **at the same time**

## Naive implementation (1/2)

- Implementation in a classical programming language (C++, Java, Python, etc.) of data structures that can represent all useful data
- Definition of ad-hoc file formats to store data on disk, with regular synchronization and a mechanism for failure recovery
- In-memory storage of application data, with data structures (search trees, hash tables) and algorithms (search, sort, aggregation, graph navigation, etc.) for efficiently finding information
- Data update functions, with code ensuring no constraint is violated

## Naive implementation (2/2)

- Definition within the application of user access rights and an authentication process; use of parallel programming to answer different requests at the same time, locks/semaphores on critical data manipulation steps
- Definition and implementation of a network communication protocol to access this software component from the Web, from a Windows application, from an accounting suite, etc.

## Naive implementation (2/2)

- Definition within the application of user access rights and an authentication process; use of parallel programming to answer different requests at the same time, locks/semaphores on critical data manipulation steps
- Definition and implementation of a network communication protocol to access this software component from the Web, from a Windows application, from an accounting suite, etc.

Lots of work!

## Naive implementation (2/2)

- Definition within the application of user access rights and an authentication process; use of parallel programming to answer different requests at the same time, locks/semaphores on critical data manipulation steps
- Definition and implementation of a network communication protocol to access this software component from the Web, from a Windows application, from an accounting suite, etc.

**Lots of work!** Requires a programmer that masters OOP, serialization, failure recovery, data structures, algorithms, integrity constraint verification, role management, parallel programming, concurrency control, network programming, etc.

## Naive implementation (2/2)

- Definition within the application of user access rights and an authentication process; use of parallel programming to answer different requests at the same time, locks/semaphores on critical data manipulation steps
- Definition and implementation of a network communication protocol to access this software component from the Web, from a Windows application, from an accounting suite, etc.

**Lots of work!** Requires a programmer that masters OOP, serialization, failure recovery, data structures, algorithms, integrity constraint verification, role management, parallel programming, concurrency control, network programming, etc. Needs to be done again **for every new 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

## Features of DBMSs (1/2)

**Physical independence.** The user of a DBMS does not need to know how data are stored (in a file, on a raw partition, in a distributed filesystem, etc.); storage can be modified without impacting data access

**Logical independence.** It is possible to provide the user with a partial view of the data, corresponding to what he needs and is allowed to access

**Ease of data access.** Use of a declarative language describing queries and updates on the data, specifying the intent of a user rather than the way this will be implemented

**Query optimization.** Queries are automatically optimized to be implemented as efficiently as possible on the database

## Features of DBMSs (2/2)

**Logical integrity.** The DBMS imposes constraints on data structure; every modification violating these constraints is denied

**Physical integrity.** The database remains in a coherent state, and data are durably preserved, even in case of software or hardware failure

**Data sharing.** Data are accessible by multiple users, concurrently, and these multiple and concurrent accesses cannot violate logical or physical data integrity

**Standardization.** The use of a DBMS is standardized, so that it may be possible to replace a DBMS with another without changing (in a major way) the code of the application



# Plan

## Introduction

Data Management

Types of DBMSs

Introduction to The Relational Model

Recursive Queries

DB or not DB?

References



## Diversity of DBMSs

- **Dozens** of existing DBMSs that are broadly used
- All DBMSs do not provide **all these features**
- DBMSs can be **differentiated** based on:
  - data model used
  - trade-offs made between performance and features
  - ease of use
  - scalability
  - internal architecture

## 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

## 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 **row by row**
- **Centralized** system, with limited distribution possibilities

**ORACLE**  
DATABASE

  
Microsoft  
SQL Server™

**IBM**  
**DB2**

**SAP** ASE

 SQLite

  
MySQL®

PostgreSQL  


## Strengths of classical relational DBMSs

- **Independence** between:
  - data model and storage structures
  - declarative queries and the way queries are executed
- **Complex** queries
- Fine **optimization** of queries, **indexes** allowing quick access to data
- **Mature** technology, **stable**, **efficient**, rich in features and interfaces
- **Integrity constraints** ensuring invariants on data
- Efficient management of **very large volume of data** (up to terabytes)
- **Transactions** (sequences of elementary operations) with guaranties on concurrency control, isolation between users, failure recovery



## ACID properties

Classical relational DBMS **transactions** satisfy **ACID** properties:

## ACID properties

Classical relational DBMS **transactions** satisfy **ACID** properties:

**Atomicity:** The set of operations within a transaction is either executed as a whole or canceled as a whole

**Consistency:** Transactions ensure integrity constraints on the base are respected

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

**Durability:** Once transactions are committed, corresponding data stay durably in the base, even in case of system failure

## Weaknesses of classical RDBMSs

- Incapable of managing **extremely large data volume** (of the order of a petabyte)
- Impossible to manage **extreme query rates** (beyond thousands of queries per second)
- The relational data model is sometimes poorly adapted to the storage and querying of **some data types** (hierarchical data, unstructured data, semi-structured data)
- ACID properties imply major **costs** in latency, disk accesses, processing time (locks, logging, etc.)
- Performances **limited by disk accesses**



# NoSQL

- **No SQL** or **Not Only SQL**
- DBMSs with other trade-offs than those made by classical systems
- **Very diverse** ecosystem
- **Desiderata**: different data model, scalability, extreme performance
- **Abandoned features**: ACID, (sometimes) complex queries



## NewSQL

- Some applications require:
  - A **rich** (SQL) query language
  - conformity to **ACID** properties
  - but **greater performance** than classical RDBMSs

# NewSQL

- Some applications require:
  - A **rich** (SQL) query language
  - conformity to **ACID** properties
  - but **greater performance** than classical RDBMSs
- Possible solutions:
  - Getting rid of classical **bottleneck** of RDBMSs: locks, logging, cache management
  - **In-memory** databases, with asynchronous copy on disk
  - **Lock-free** concurrency control (MVCC)
  - **Shared-nothing** distributed architecture with transparent **load balancing**





# Plan

Introduction

Introduction to The Relational Model

Model

Relational Algebra

SQL

Relational Calculus

Recursive Queries

DB or not DB?

References

## 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  $(L_i, \tau_i)$  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 (ISO 8601 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 note  $t[A]$  (or  $t[L]$ ) the value of the  $i$ th component of  $t$ .
- Similarly, if  $\mathcal{A}$  is a  $k$ -tuple of attributes among the  $n$  attributes of  $R$ ,  $t[\mathcal{A}]$  is the  $k$ -tuple formed from  $t$  by concatenating the  $t[A]$  for  $A$  in  $\mathcal{A}$ .
- A **tuple** is an  $n$ -tuple for some  $n$ .

## Simple integrity constraints

One can add to the relational schema some **integrity constraints**, of different nature, to define a notion of **validity** of an instance

- **Key**. A tuple of attribute  $\mathcal{A}$  of a relation schema  $R$  is a **key** if there cannot exist two distinct tuples  $t_1$  and  $t_2$  in an instance of  $R$  such that  $t_1[\mathcal{A}] = t_2[\mathcal{A}]$
- **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 schema  $S$  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 each 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 a “@”
- 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 cannot be occupied twice the same night, which depends on the date and the number of nights for multiple tuples of Reservation)

## Variants: named and unnamed perspectives

The version presented considers the attributes of a relation are ordered and have a name. This is what best matches the way RDBMSs work, but not necessarily the most pleasant to reason on the relational model.

**Named perspective.** We forget the position of attributes, and consider they are uniquely identified by their names.

**Unnamed perspective.** We forget the name of attributes, and consider they are uniquely identified by their position. One uses notation such as  $t[2]$  to access the value of the second attribute of a tuple.

No major impact, one will use one or the other depending on what is convenient.

## 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, but explain where differences matter

## Variant: untyped version

- In implementations, attributes are **always typed**
- In models and theoretical results, one often abstracts attribute types away and considers each attribute has a **universal type**  $\mathcal{V}$
- We will most often omit **attribute types**



# Plan

Introduction

Introduction to The Relational Model

Model

Relational Algebra

SQL

Relational Calculus

Recursive Queries

DB or not DB?

References

## 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	$A_1 \dots A_n \in \mathcal{L}$
$\sigma_\varphi$	1	Selection	$\varphi$ formula
$\times$	2	Cross product	
$\cup$	2	Union	
$\setminus$	2	Difference	
$\bowtie_\varphi$	2	Join	$\varphi$ 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:  $\text{Reservation} \bowtie_{\text{guest}=\text{id}} \text{Guest}$

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 database schema
- For example:
  - No reference to the name of a relation that doesn't exist in the database schema
  - One cannot reference (within a renaming, projection, selection, join) an attribute that does not exist in the result of a sub-expression
  - One cannot union two relations with different attributes
  - One cannot produce (cross product, join, renaming) a table with two attributes with the same name
- Systems implementing the relational algebra may do a static or dynamic verification 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]:

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

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

room	avg
302	6
504	3.5



# Plan

Introduction

Introduction to The Relational Model

Model

Relational Algebra

SQL

Relational Calculus

Recursive Queries

DB or not DB?

References

## SQL

- **Structured Query Language**, standard 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 next
- Many little things (e.g., available types) vary between RDBMSs instead of following the standard
- Differences more **syntactical** than major
- Where it makes a difference, we use the **PostgreSQL** version
- Two main parts: DDL (**Data Definition Language**) to define the schema and DML (**Data Manipulation Language**) to query and update data
- **Declarative** language: express what you mean, the system will take care of translating this into an **efficient execution plan**



## Syntax of SQL

- Quite **verbose**, designed to be almost readable as English words [Chamberlin and Boyce, 1974]
- Keywords are **case-insensitive**, traditionally written in all uppercase
- Identifiers often **case-insensitive** (depends of the RDBMS), often written with an initial uppercase for table names, in all lower case for attribute names
- **Comments** introduced with --
- SQL statements end with a “;” in some contexts (e.g., command line client) but the “;” is not properly part of the statement

## NULL

- In SQL, NULL is a special value that an attribute can take within a tuple
- Denotes **absence of value**
- Different from 0, from an empty string, etc.
- Weird **tri-valued** logic: True, False, NULL
- A normal comparison (equality, inequality, etc.) with NULL always returns NULL
- **IS NULL, IS NOT NULL** can be used to test whether a value is NULL
- NULL est ultimately converted to 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 destroy 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

Specified at the creation of a table, or added later on (with **ALTER TABLE**)

**PRIMARY KEY** for the primary key; only one per table, it is the 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 take the NULL value

## Constraints

```
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)
);
```

## 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;
```

# Updates

## 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

General following form:

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

**SELECT** projection, renaming, aggregation

**FROM** cross product

**WHERE** selection (optional)

**GROUP BY** grouping (optional)

**HAVING** selection on the grouping (optional)

**UNION** union (optional)

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



## 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

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

```

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

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

```
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		
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

Reservation  $\bowtie_{\text{guest=id}}$  Guest

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

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

```

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

```

# Plan

## Introduction

### Introduction to The Relational Model

Model

Relational Algebra

SQL

Relational Calculus

## Recursive Queries

## DB or not DB?

## References

## Relational calculus

- **Logical language** to express queries
- **First-order logic** formula, without function symbols, and with **relation symbols** the labels of the database schema (plus comparison predicates)
- **Unnamed, untyped** perspective
- Fix:
  - A set  $\mathcal{X}$  of variables
  - A set  $\mathcal{V}$  of values
  - A database schema  $S$

## Relational calculus: Syntax

- For every relation  $R \in S$  of arity  $n$ , for every  $(\alpha_1, \dots, \alpha_n) \in (\mathcal{X} \cup \mathcal{V})^n$ :  $R(\alpha_1, \dots, \alpha_n) \in \text{FO}$
- Also allow equality predicate, possibly inequality
- For every  $(\varphi_1, \varphi_2) \in \text{FO}^2$ , for every  $x \in \mathcal{X}$ :
  - $\varphi_1 \wedge \varphi_2 \in \text{FO}$
  - $\varphi_1 \vee \varphi_2 \in \text{FO}$
  - $\neg \varphi_1 \in \text{FO}$
  - $\forall x \varphi_1 \in \text{FO}$
  - $\exists x \varphi_1 \in \text{FO}$
- **Free variables** of  $\varphi \in \text{FO}$ : variables  $x$  appearing in  $\varphi$  and not qualified by a  $\forall x$  or a  $\exists x$
- One writes a relational calculus **query** in the form  $Q(x_1, \dots, x_m) = \varphi$  where  $x_1, \dots, x_m$  are free variables of  $\varphi$

## Relational calculus: Semantics

- A relational calculus query on schema  $S$  can be seen as a **function** with input a database  $D$  over  $S$  and producing a relation as output
- $\text{adom}(D)$ : **active domain** of  $D$ , set of values in  $D$
- If  $Q(x_1, \dots, x_n) = \varphi$  is a calculus query over  $S$  and  $D$  a database over  $S$ , then:

$$Q(D) = \{ (v_1, \dots, v_n) \in (\text{adom}(D))^n \mid D \models \varphi[x_1/v_1, \dots, x_n/v_n] \}$$

where  $D \models \varphi$  is defined inductively:

- $D \models R(u_1, \dots, u_m) \iff R(u_1, \dots, u_m) \in D$
- $D \models \varphi_1 \wedge \varphi_2 \iff D \models \varphi_1 \wedge D \models \varphi_2$
- $D \models \varphi_1 \vee \varphi_2 \iff D \models \varphi_1 \vee D \models \varphi_2$
- $D \models \neg \varphi_1 \iff D \not\models \varphi_1$
- $D \models \forall x \varphi_1 \iff \forall v \in \text{adom}(D) D \models \varphi_1[x/v]$
- $D \models \exists x \varphi_1 \iff \exists v \in \text{adom}(D) D \models \varphi_1[x/v]$

## Codd's theorem

### Theorem ([Codd, 1972])

*The relational algebra and the relational calculus are equivalent:*

- *for every relational algebra query  $q$  over a schema  $S$ , there exists a relational calculus query  $Q$  over  $S$  such that for every database  $D$  over  $S$ ,  $q(D) = Q(D)$*
- *for every relational calculus query  $Q$  over a schema  $S$ , there exists a relational algebra query  $q$  over  $S$  such that for every database  $D$  over  $S$ ,  $q(D) = Q(D)$*

*Furthermore, translating from one formalism to the other can be done in polynomial time.*

## Why is this important?

- Allows using a **declarative formalism** to specify queries: logics... or SQL
- These queries are then compiled via Codd's transformation into an **algebraic formalism**
- Algebraic queries are then **optimized**, by using the properties of the relational algebra (transformation rules, e.g., pushing selection within joins, exploiting associativity of joins, etc.)
- Optimized queries can then be **evaluated**, by exploiting the fact that each operator of the relational algebra can easily be implemented (in several different ways, to be chosen based on a cost function)
- This is RDBMS Implementation 101, a main reason of the success of RDBMSs!

## Subclasses of queries

- **Conjunctive query (CQ):** relational calculus query without  $\vee, \neg, \forall$
- **Positive query (PQ):** relational calculus query without  $\neg, \forall$
- **Union of conjunctive queries (UCQ):** special case of positive query where the  $\vee$  and  $\wedge$  form a DNF formula

## Subclasses of queries

- **Conjunctive query (CQ)**: relational calculus query without  $\vee, \neg, \forall$
- **Positive query (PQ)**: relational calculus query without  $\neg, \forall$
- **Union of conjunctive queries (UCQ)**: special case of positive query where the  $\vee$  and  $\wedge$  form a DNF formula

### Expressiveness

- CQs are **equivalent** to the relational algebra without  $\cup$  and  $\setminus$ , and where  $\sigma$  does not feature disjunction
- UCQs are **equivalent** to PQs (but exponential blow-up), and equivalent to the relational algebra without  $\setminus$



# Plan

Introduction

Introduction to The Relational Model

Recursive Queries

Datalog

Recursion in SQL

DB or not DB?

References

## Motivation: recursive queries

- Query languages considered so far (relational algebra, calculus) have a **limited horizon**
- Some data structures (trees, graphs) require **arbitrarily deep** navigation, **recursion**
- How can we build a theory of **recursive query languages**?
- RDBMSs are **not always** adapted to this type of data/queries, cf. XML or graph DBMSs
- **Example application**: transitive closure of a graph  $G(\textit{from}, \textit{to})$

# Datalog

- Simplest recursive query language: adding recursion to **conjunctive queries**
- Inspired from **logic programming**
- Datalog query (or **program**): set of rules that produce **intensional facts**
- **Schema** of a Datalog program: classical database schema (**extensional schema**) + (disjoint) schema of intensional facts (**intensional schema**)
- Fix a distinguished relation *Goal* of the intensional schema, whose arity is the arity of the query

## Syntax

Finite set of rules  $r$  of the form:

$$\underbrace{S(\mathbf{y})}_{\text{head}} \leftarrow \underbrace{R_1(\mathbf{x}_1), \dots, R_n(\mathbf{x}_n)}_{\text{body}}$$

with:

- $S$  relation of the **intensional** schema
- $R_1, \dots, R_n$  **relations** of the intensional or extensional schemas
- $\mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{y}$ : tuples of **variables** (or possibly constants), of arity compatible with the relations
- Each variable in the head is **present** in the body

## Fix-point semantics

- Each rule  $r$  of a program  $P$  can be seen as a **conjunctive query** on the database  $D$ :

$$r(D) := \{S(\mathbf{y}) \mid \exists z_1 \dots z_k R_1(\mathbf{x}_1) \in D \wedge \dots \wedge R_n(\mathbf{x}_n) \in D\}$$

where the  $z_i$ 's are the variables of the rule body

- Consequence operator**  $\Gamma_P$  defined by:

$$\Gamma_P(D) := D \cup \bigcup_{r \in P} \{r(D)\}$$

- We consider the **sequence**  $(D_n)$  defined by:  
 $D_0 = D, D_{n+1} = \Gamma_P(D_n)$
- The semantics of  $P$  over  $D$  is the set of facts of the relation *Goal* in  $D_\infty$ , the **fixpoint** of the sequence  $(D_n)$



## Example: transitive closure

$$Goal(x, y) \leftarrow G(x, y)$$

$$Goal(x, y) \leftarrow Goal(x, z), G(z, y)$$



# Plan

Introduction

Introduction to The Relational Model

Recursive Queries

Datalog

Recursion in SQL

DB or not DB?

References

## Common Table Expressions (CTE)

- Only way in SQL to introduce recursion (also called **hierarchical queries**)
- **Idea:** We declare a table, whose definition refers to itself
- Inspired from **inflationary fixpoint logics** but also from recursive functions in programming languages

## Syntax

```

WITH RECURSIVE T(x1, ..., xn) AS (
  SELECT -- base case
UNION
  SELECT -- recursive case, using table T
)
SELECT * FROM T;

```

This precise form, using **UNION** (or **UNION ALL**) between base and recursive cases is **compulsory**

## Semantics

- The query is evaluated **iteratively**, from  $T = \emptyset$  and by replacing at every step  $T$  with the result of evaluating the definition of  $T$
- This terminates when a **fixpoint is reached**
- The query defining  $T$  must be **monotone** ( $T$  can only grow at each step)
- Optimizations possible (and used) in order to only take into account **new facts** at every step (thanks to monotonicity)
- Infinite loops **possible** if new values are created

## Example: transitive closure

```

WITH RECURSIVE C(f,t) AS (
  SELECT * FROM G
  UNION
  SELECT C.f, G.t FROM C JOIN G ON C.t=G.f
)
SELECT * FROM C;

```



## Notes on historical support

- CTE normalized **late** [ISO, 1999] in SQL
- **Non-compatible** historical extensions: (**CONNECT BY** introduced by Oracle, reused by some other DBMSs)
- Support is now **correct**
- Recursive queries are not a priority of DBMSs, poor optimization (in PostgreSQL, recursive CTEs are an **optimization fence**)



# Plan

Introduction

Introduction to The Relational Model

Recursive Queries

**DB or not DB?**

Is This Necessary?

pandas

Unix command line tools

References



## Is a DBMS necessary?

- DBMSs are good at managing very large amounts of data



## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**... but sometimes data we deal with are **not that large!**



## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**. . . but sometimes data we deal with are **not that large!**
- DBMSs provide a standard **declarative** query language with smart query **optimization**

## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**... but sometimes data we deal with are **not that large!**
- DBMSs provide a standard **declarative** query language with smart query **optimization**... but sometimes it is fine to describe **precisely how** the computation is to be performed!

## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**. . . but sometimes data we deal with are **not that large!**
- DBMSs provide a standard **declarative** query language with smart query **optimization**. . . but sometimes it is fine to describe **precisely how** the computation is to be performed!
- DBMSs provide different views of data, **isolation between users**, concurrency control

## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**. . . but sometimes data we deal with are **not that large!**
- DBMSs provide a standard **declarative** query language with smart query **optimization**. . . but sometimes it is fine to describe **precisely how** the computation is to be performed!
- DBMSs provide different views of data, **isolation between users**, concurrency control. . . but sometimes there is **only a single user**



## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**. . . but sometimes data we deal with are **not that large!**
- DBMSs provide a standard **declarative** query language with smart query **optimization**. . . but sometimes it is fine to describe **precisely how** the computation is to be performed!
- DBMSs provide different views of data, **isolation between users**, concurrency control. . . but sometimes there is **only a single user**
- DBMSs provide updating features, ensure constraints are not violated while updating, ensure updated data is in a consistent state



## Is a DBMS necessary?

- DBMSs are good at managing very **large amounts of data**. . . but sometimes data we deal with are **not that large!**
- DBMSs provide a standard **declarative** query language with smart query **optimization**. . . but sometimes it is fine to describe **precisely how** the computation is to be performed!
- DBMSs provide different views of data, **isolation between users**, concurrency control. . . but sometimes there is **only a single user**
- DBMSs provide updating features, ensure constraints are not violated while updating, ensure updated data is in a consistent state. . . but sometimes a dataset **is not to be modified**

## Possible alternatives

- In memory** Ad-hoc management of data stored in main memory, within some programming language – if the data fits within memory. Will be illustrated by the **pandas** Python library.
- On disk** Ad-hoc management of data on disk, stored in files, either through programming or through the use of external tools. Will be illustrated by the **Unix command line** tools.

## When a DBMS is necessary

- When the data needs to be used by **other applications**
- When data **updating**, transactions, concurrency control, user isolation, etc., is important
- When queries become **complex**, and are more manageable and easily optimized in a declarative query language like SQL than in an ad-hoc language
- When data volumes are **too large** for simple in-memory storage or for ad-hoc disk accesses, when indexes are required



## Data management concepts

Even when not using a DBMS, data management concepts are important:

- expressing operations in terms of **formal operators** such as selections, projections, joins, etc., allows to **better understand and describe** what needs to be performed
- paying attention to integrity constraints allows catching up potential **errors in data formats**
- the notion of **physical and logical independence** may still be relevant in how to design a computation



## Assumption

- Data still follows the relational data model (similar processes may be followed for other data models)
- Data will be stored in simple text files with newline-separated rows, and delimiter-separated attribute values on each row
- Data available in extension as files, one per table



## pandas

- Rich library for expressing **complex manipulation of tabular data** in Python
- Data tables available in Python in the form of a **DataFrame object**
- Heavily inspired by the way data tables are handled in statistical computing language such as **R** and **SAS**
- **Not** a **declarative** language: the way an expression is written is the way it will be executed (with minor optimizations)
- Results in code **less verbose** than SQL, but also somewhat **more cryptic**
- All data is stored **in main memory**: does not scale to large datasets
- As this is Python code, **arbitrary** code can be written, interfaces with other libraries (e.g., for deep learning)

## The DataFrame object

- Representation of a **relation** (tabular data, fixed number of columns/attributes, names for attributes, etc.)
- Each **row** can be assigned an **index**, i.e., a name; similar concept to that of primary key – if no name assigned, rows are referred to by a sequential numbering
- Relies on Series objects, which are unidimensional arrays of data; each column is such a series
- The DataFrame is said to have two axes: axis 0 is the rows, axis 1 the columns
- As there are row indexes, there are **column indexes**, i.e., attribute names

## Constructing a DataFrame

```
import pandas as pd
```

```
# Literal DataFrame
```

```
guest = pd.DataFrame(  
    data={  
        'name': ['John Smith', 'Alice Black', 'John Smith'],  
        'email': ['john.smith@gmail.com',  
                 'alice@black.name',  
                 'john.smith@ens.fr']  
    },  
    index = pd.Index([1, 2, 3], name='id'))
```

```
# Read DataFrame from CSV file, first column as row index  
reservation = pd.read_csv('reservation.csv', index_col=0)
```

## Renaming (1/2)

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	

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

```
guest.index.name='guest'
```

(This changes the guest DataFrame)



## Renaming (2/2)

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

$$\rho_{\text{email} \rightarrow \text{e-mail}}(\text{Guest})$$

```
guest.rename(columns={'email': 'e-mail'})
```



## Projection (1/2)

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

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

```
guest[['email']]
```

(The row index always comes first)



## Projection (2/2)

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

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

```
guest.reset_index(['email'])
```

(A new default index is generated)

```
guest.set_index('email')[[]]
```

(The email column becomes the index)



## 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

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

```
reservation[(reservation.arrival > '2017-01-12') & \
            (reservation['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

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

```
guest[[]].reset_index().merge(\
    guest[['name']], how='cross').\
    drop_duplicates().sort_values(['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

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

```
pd.concat([
    reservation[reservation.guest==2].\
        reset_index()[['room']],
    reservation[reservation.arrival=='2017-01-15'].\
        reset_index()[['room']]
]).drop_duplicates()
```

## 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

$$\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation}))$$

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

```
r1=reservation[reservation.guest==2].\
  reset_index()[['room']]
r2=reservation[reservation.arrival=='2017-01-15'].\
  reset_index()[['room']]
r1.merge(r2,how='outer',indicator=True).\
  query('_merge=="left_only"')[['room']].\
  drop_duplicates()
```



## 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  $\bowtie_{\text{guest=id}}$  Guest

```
pd.merge(reservation,
         guest,
         left_on='guest',
         right_on='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

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

```
reservation.groupby('room')[['nights']].\
  mean().\
  query('nights>3').\
  sort_values(by='room')
```

## But also

`df.sort_values` to order results (similar to **ORDER BY** in SQL)

`df.head` to limit the number of answers (similar to **LIMIT** in SQL)

`df.tail` to only keep the last answers

`df.size` to count the number of elements

Recursion requires iterating in Python



# Plan

Introduction

Introduction to The Relational Model

Recursive Queries

**DB or not DB?**

Is This Necessary?

pandas

**Unix command line tools**

References



## Command line tools

- **Classical tools** used within a Unix/Linux/WSL shell to manipulate text files
- **Disk-based** and **pipelined** file manipulation: usually no important use of memory
- **Scales** to very large files, but no indexing capabilities, only linear (and pipelined) processing of files

## Pipelines

- If  $a$  and  $b$  are two commands, then  $a | b$ :
  - simultaneously launches commands  $a$  and  $b$
  - **redirects the standard output** of  $a$  to the **standard input** of  $b$
- This means  $b$  is provided (on its standard input) the standard output of  $a$ , **as it is produced**
- **Blocking** (with buffering): if  $b$  stops reading its input,  $a$  stops producing an output (and conversely)
- Central point of Unix/Linux command line philosophy, very convenient
- Can be chained:  $a | b | c | d | \dots$

## Reading from standard input or arguments

- Most common Unix commands can read their input:
  - either through their **standard input** (useful for pipelines)
  - or within a **file** given as **argument**
- **Process substitution**:  $a <(b)$  launches command  $b$  and provides to command  $a$  a filename (actually a **temporary named pipe**) which, if read, provides the output of the  $b$  command
- Often, when a filename is expected as argument,  $-$  means to read from standard input instead

## Getting help on a command

- Try *command* --help or *command* -h
- **man** is a documentation integrated within Unix/Linux:  
“man *command*” displays a manual page about a command
- **man -k** searches a man entry by keyword
- **whatis** short summary of a command

## Format of input files

- Rows separated by newlines
- Attributes separated by a special delimiter character, “,” in examples; commands use a flag (-d, -t, -F) to indicate the delimiter
- No header line; can be removed using `tail -n +2 file.csv`
- More complex CSV files: see the [csvkit](#) or [csvquote](#) tools

## awk

- Programming language dedicated to the processing of tabular data within text files
- General form of an awk program:
 

```
BEGIN { instructions1 }
condition { instructions2 }
END { instructions3 }
```
- instructions1 is executed at the beginning of a file, the **BEGIN** block can be omitted
- condition is a condition for each line to be processed; if omitted, defaults to matching every line
- instructions2 is executed at each line matched by conditions; if omitted defaults to print the line
- instructions3 is executed at the end of a file, the **END** block can be omitted

# 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

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

```
awk -F, '{print $3 ", " $1}' guest.csv
```

or

```
cut -d, -f3,1 guest.csv
```

(cut doesn't allow reordering or repetition of columns)



## 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

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

```
awk -F, '$4 > "2017-01-12" && $2 == 2' reservation.csv
```

# 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

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

```
join -t, -j 2 -o '1.1,2.1' \  
  <(cut -d, -f1 guest.csv) \  
  <(cut -d, -f2 guest.csv) | \  
  sort -t, -k2,1 | uniq
```



## Union

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

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

```
cat <(awk -F, '$2==2 {print $3}' reservation.csv) \
  <(awk -F, '$4=="2017-01-15" {print $3}' reservation.csv) | \
  sort -u
```



## 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

$$\Pi_{\text{room}}(\sigma_{\text{guest}=2}(\text{Reservation}))$$

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

```
join -v2 \
  <(awk -F, '$2==2 {print $3}' reservation.csv|sort) \
  <(awk -F, '$4=="2017-01-15" {print $3}' reservation.csv|sort)
```

# 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  $\bowtie_{\text{guest=id}}$  Guest

```
sort -t, -k2 reservation.csv | \
  join -t, -1 2 -2 1 - <(sort guest.csv)
```



## 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

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

```
awk -F, '{s[$3]+=$5;++c[$3]}
END {for (r in s) print r "," s[r]/c[r]} ' \
reservation.csv | \
awk -F, '$2>3' | sort -t, -n
```

## But also

`sort` to order results (similar to **ORDER BY** in SQL)

`head` to limit the number of answers (similar to **LIMIT** in SQL)

`tail` to only keep the last answers

`tac` to reverse the order

`paste` to merge lines of several inputs, in order ( $n$ th lines of each file are merged)

`wc -l` to count the number of lines

Recursion requires looping, e.g., with a **while** loop [Rebele et al., 2018]



# Plan

Introduction

Introduction to The Relational Model

Recursive Queries

DB or not DB?

References



## References (1/2)

- **Basics** on what data management research is [Benedikt and Senellart, 2012]
- Classic textbook on the **foundations** of data management, database theory [Abiteboul et al., 1995]
- Modern textbook, being developed, on **database theory** [Arenas et al., 2022]
- Classic textbook on **database systems** [Garcia-Molina et al., 2008]

## References (2/2)

- **Details of SQL:** standards are not public documents (and not useful for a final user); use the RDBMS documentation instead
- For example, **PostgreSQL** has a comprehensive documentation at <https://www.postgresql.org/docs/> (and using \h in the command line client)
- **pandas documentation:**  
<https://pandas.pydata.org/docs/>
- Good introduction to the Unix/Linux command line:  
[William E. Shotts, 2019]

## Bibliography I

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

Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. Database theory.  
<https://github.com/pdm-book/community>, 2022.  
Preliminary version.

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

- Edgar F. Codd. Relational completeness of data base sublanguages. In *Data Base Systems. Courant Computer Science Symposium*, 1972.
- Hector Garcia-Molina, Jeffrey D. Ulman, and Jennifer Widom. Pearson, 2008.
- 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.

## Bibliography III

- Thomas Rebele, Thomas Pellissier Tanon, and Fabian M. Suchanek. Bash datalog: Answering datalog queries with unix shell commands. In Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez- Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, volume 11136 of *Lecture Notes in Computer Science*, pages 566–582. Springer, 2018. doi: 10.1007/978-3-030-00671-6\\_33. URL [https://doi.org/10.1007/978-3-030-00671-6\\_33](https://doi.org/10.1007/978-3-030-00671-6_33).
- Jr. William E. Shotts. *The Linux Command Line: A Complete Introduction*. No Starch Press, 2nd edition, 2019.