# Relational Database Management

## Advanced Databases

Pierre Senellart

ÉCOLE NORMALE SUPÉRIEURE | PSL

22 September 2022

# Plan

# 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

# 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

NoSQL:

**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

# 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

# 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

Google Cloud Spanner Clustrix VOLT ACTIVE DATA

# Plan

# 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 $(A_1, \ldots, A_n)$ 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`: $((\mathtt{id}, \mathtt{INTEGER}), (\mathtt{name}, \mathtt{TEXT}), (\mathtt{email}, \mathtt{TEXT}))$
- `Reservation`: $((\mathtt{id}, \mathtt{INTEGER}), (\mathtt{guest}, \mathtt{INTEGER}), (\mathtt{room}, \mathtt{INTEGER}), (\mathtt{arrival}, \mathtt{DATE}), (\mathtt{nights}, \mathtt{INTEGER}))$

# Database

## Definition

An instance of a relation schema $((L_1, \tau_1), \ldots, (L_n, \tau_n))$ (also called a relation on this schema) is a finite set $\{t_1, \ldots, t_k\}$ of tuples of the form $t_j = (v_{j1}, \ldots, v_{jn})$ 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

# 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 | $R \in \mathcal{L}$ |
| $\rho_{A \to B}$ | 1 | Renaming | $A, B \in \mathcal{L}$ |
| $\Pi_{A_1 \ldots A_n}$ | 1 | Projection | $A_1 \ldots A_n \in \mathcal{L}$ |
| $\sigma_\varphi$ | 1 | Selection | $\varphi$ formula |
| $\times$ | 2 | Cross product | |
| $\cup$ | 2 | Union | |
| $\setminus$ | 2 | Difference | |
| $\bowtie_\varphi$ | 2 | Join | $\varphi$ formula |

# Relation name

Guest

| 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: `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**

| 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: $\rho_{\texttt{id}\to\texttt{guest}}(\texttt{Guest})$

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

Result:

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

# Selection

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

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

Result:

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

# 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 |

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

Result:

| room |
|---|
| 107 |
| 302 |
| 504 |

# 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 |

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

**Result:**

| room |
|---|
| 107 |
| 302 |
| 504 |

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

# Difference

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

Result:

| room |
|---|
| 107 |

# Difference

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

Result:

| room |
|---|
| 107 |

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

# 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 |

Expression: $\texttt{Reservation} \bowtie_{\texttt{guest}=\texttt{id}} \texttt{Guest}$

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:

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

- 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]:

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

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

# 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 ";" id 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 |

$$\rho_{\texttt{id}\to\texttt{guest}}(\texttt{Guest})$$

```sql
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 |

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

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

# Selection

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 |

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

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

# 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 |

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

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

# 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 |

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

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

# Difference

**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 |

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

```sql
  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 |

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

```sql
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

| 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 |

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

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

# Plan

# 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 $(\alpha_1, \ldots, \alpha_n) \in (\mathcal{X} \cup \mathcal{V})^n$: $R(\alpha_1, \ldots, \alpha_n) \in \text{FO}$
- 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 $Q(x_1, \ldots, x_m) = \varphi$ where $x_1, \ldots, x_m$ 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
- adom($D$): active domain of $D$, set of values in $D$
- If $Q(x_1, \ldots, x_n) = \varphi$ is a calculus query over $S$ and $D$ a database over $S$, then:

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

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

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

# 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

# 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 $G(\mathit{from}, \mathit{to})$

# 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:

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

with:

- $S$ relation of the intensional schema
- $R_1, \ldots, R_n$ relations of the intensional or extensional schemas
- $\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}$: 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

# While operator

- Algebra: essentially imperative programming (in contrast to calculus)
- One can add to the algebra:
  - the possibility to define intermediary variables and to assign them values (does not affect expressive power)
  - a **while** operator of the form:

**while change do**

*...assign value to one or several variables...*

**done**

- Semantics: the content of the loop is evaluated while the assignments change the underlying variables
- Infinite loops possible!

# Inflationary vs non-inflationary while

- Two variants:

  Non-inflationary: Assignment operator :=, arbitrary assignment

  Inflationary: Assignment operator +=, the assigned value can only grow
- Infinite loops impossible with an inflationary **while** (because the active domain is finite)
- Non-inflationary potentially more expressive

# Example: transitive closure

We introduce a relation schema $C$ with attributes *from* and *to*, similarly to $G$.

Non-inflationary

$C := G$

**while change do**

$C := C \cup \pi_{from,to}(\rho_{to\to int}(C) \bowtie \rho_{from\to int}(G))$

**done**

$C$

Inflationary

$C \mathrel{+}= G$

**while change do**

$C \mathrel{+}= \pi_{from,to}(\rho_{to\to int}(C) \bowtie \rho_{from\to int}(G))$

**done**

$C$

# Plan

# Non-inflationary fixpoint

- We add to the relational calculus a fixpoint construction
- Let $\varphi(T)$ be a calculus formula, mentioning the schema relations as well as a new relation $T$, having for free variables $x_1, \dots, x_n$
- Then $\mu_T[\varphi(T)](x_1, \dots, x_n)$ is a formula of the fixpoint calculus
- Semantics: in terms of the least fixpoint (if it exists) of the relation $T$, obtained by replacing $T$ at each step with the set of facts of the form $T(x_1, \dots, x_n)$ for $\varphi(T)(x_1, \dots, x_n)$ satisfied (starting with $T = \emptyset$)
- Equivalent to algebra with non-inflationary **while**!

# Inflationary fixpoint

- We add to the relational calculus a fixpoint construction
- Let $\varphi(T)$ be a calculus formula, mentioning the schema relations as well as a new relation $T$, having for free variables $x_1, \ldots, x_n$
- Then $\mu^{+}_{T}[\varphi(T)](x_1, \ldots, x_n)$ is a formula of the fixpoint calculus
- Semantics: in terms of the least fixpoint of the relation $T$, obtained by adding to $T$ at each step with the set of facts of the form $T(x_1, \ldots, x_n)$ for $\varphi(T)(x_1, \ldots, x_n)$ satisfied (starting with $T = \emptyset$)
- Equivalent to algebra with inflationary **while**!

# Example: transitive closure

Non-inflationary

$$\{(x,y) \mid \mu_C [G(x,y) \lor C(x,y) \lor (\exists z\, C(x,z) \land G(z,y))](x,y)\}$$

Inflationary

$$\{(x,y) \mid \mu^+_C [G(x,y) \lor (\exists z\, C(x,z) \land G(z,y))](x,y)\}$$

# Query evaluation

- Query $Q$ in some query language (CQ, FO, FO+$\mu$, FO+$\mu^+ \ldots$) – we will use a logical formalism here
- Database $D$ (always finite!)
- Query evaluation: Computing $Q(D)$
- Complexity of this problem?
- To simplify the study of complexity, we often assume that $Q$ is a Boolean query, i.e., it returns $\top$ or $\bot$

# Data complexity

For some fixed $Q$, what is the complexity of computing $Q(D)$ in terms of the size of the database $D$?

# Combined complexity

For some query language $\mathcal{Q}$, what is the complexity of computing $Q(D)$ in terms of the size of the query $Q \in \mathcal{Q}$ and of the database $D$?

# Complexity classes

- We restrict to Boolean problems (returning $\top$ or $\bot$)
- Set of all problems solvable by a resource-constrained computing method:
- For example:
  - PTIME: deterministic Turing machine in polynomial time
  - NP: non-deterministic Turing machine in polynomial time
  - PSPACE: deterministic Turing machine in polynomial space
  - $\mathrm{AC}^0$: Boolean circuit of polynomial size and constant depth
- We know that: $\mathrm{AC}^0 \subsetneq \mathrm{PTIME} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}$
- Open whether $\mathrm{PSPACE} \subseteq \mathrm{PTIME}$ (!)

# Membership and hardness for a class

- A problem $P$ belongs to a complexity class $\mathcal{C}$ (or in $\mathcal{C}$) if it is solvable by the corresponding resource-constrained computing method
- A problem $P$ is hard for a complexity class $\mathcal{C}$ (or $\mathcal{C}$-hard) if there exists a reduction that transforms whatever problem $P' \in \mathcal{C}$ into an instance of the problem $P$
- complete: in $\mathcal{C}$ + $\mathcal{C}$-hard
- Several ways to define reductions
- Here, we assume that there exists a function computable in polynomial time that transforms one instance $I'$ of problem $P'$ into an instance $I$ of $P$ such that $P(I) = P'(I')$

# Descriptive complexity

- A query language $\mathcal{Q}$ captures a complexity class $\mathcal{C}$ if:
  - For all $Q \in \mathcal{Q}$, query evaluation of query $Q$ in in $\mathcal{C}$ (data complexity)
  - For all problem $P$ in $\mathcal{C}$, there exists a query $Q \in \mathcal{Q}$ such that evaluating $Q$ exactly solves $P$ (without a reduction)!
- If $\mathcal{Q}$ captures $\mathcal{C}$ and if $\mathcal{C}$ has problems that are complete for $\mathcal{C}$, then there exists $Q \in \mathcal{Q}$ such that $Q$ is $\mathcal{C}$-complete, but the converse is not true

# Plan

# Data complexity

**Theorem**

*CQ evaluation is PTIME in data complexity.*

**Proof.**

By enumerating all valuations of variables of the query in the database. We will see later a much stronger result. □

# Combined complexity

**Theorem**

*CQ evaluation is NP-complete in combined complexity.*

**Proof.**

Membership in NP is easy. Hardness for NP can be proved by reduction from graph 3-colorability. □

# $\alpha$-acyclic query

- A CQ can be seen as a hypergraph (vertices are variables, hyperedges the atoms of the CQ, labeled by the relation name)
- A hypergraph $\mathcal{H}$ has a join tree where one can find a tree whose nodes are labeled by the hyperedges of $\mathcal{H}$ and such that:
  - every hyperedge of $\mathcal{H}$ appears as the label of one node of the tree;
  - for every vertex $x$ of $\mathcal{H}$, the set of tree nodes labeled by a hyperedge referring to $x$ is a connected subtree
- A query is $\alpha$-acyclic if its hypergraph has a join tree
- Can be obtained in linear time if it exists [Tarjan and Yannakakis, 1984]

# Yannakakis's algorithm [Yannakakis, 1981]

- Algorithm to evaluate acyclic queries (non-necessarily Boolean):
  1. Construct the join tree
  2. Eliminate all useless tuples of a relation with the semijoin operator $\ltimes$: $R \ltimes S = \Pi_{R.*}(R \bowtie S)$ by navigating twice in the join tree: from bottom up, then from top down
  3. Evaluate the query bottom up, by computing joins following the tree and by projecting useless variables out as you go
- Polynomial complexity in the size of the query, the input, and the output (combined complexity)

# Plan

# Data complexity

**Theorem**

*FO evaluation is PTIME in data complexity.*

**Proof.**

By rewriting in prenex form and naive evaluation. □

# FO does not capture the whole of PTIME

**Theorem**

*One cannot compute in FO that a relation containing a total order has an even number of elements, or that a graph is connected.*

Fairly complex to prove, relies on Ehrenfeucht–Fraïssé games (see [Libkin, 2004]).

# Data complexity, more precise

**Theorem**

*FO evaluation is $AC^0$ in data complexity.*

**Proof.**

By rewriting to the relational algebra. □

# Combined complexity

**Theorem**

*FO evaluation is PSPACE-complete in combined complexity.*

**Proof.**

Membership in PSPACE easy. Hardness for PSPACE from the QSAT problem. □

# Plan

# Data complexity

**Theorem**

*Evaluation of FO+$\mu^+$ is PTIME in data complexity.*
*Evaluation of FO+$\mu$ is PSPACE in data complexity.*

**Proof.**

Direct. Only need polynomial space to detect infinite loops. □

# Connectedness, parity

### Theorem

*Connectedness is expressible in FO+$\mu^+$.*

*The problem of determining whether an arbitrary relation has an even number of elements (or tuples) cannot be expressed in FO+$\mu$.*

### Proof.

First part easy, recall transitive closure query.
Second part shown in [Abiteboul et al., 1995], chapter 17. □

# Parity and FO+$\mu^+$, with order

**Theorem**

*FO+$\mu^+$ can compute if a relation that contains a total order has an even number of elements.*

**Proof.**

Explicit construction. □

# More general results

**Theorem**

*$FO+\mu^+$ captures PTIME on databases including a total order relation of the domain.*

*$FO+\mu$ captures PSPACE on databases including a total order relation of the domain.*

**Proof.**

Simulation of polynomial-time or polynomial-space deterministic Turing machines with a fixpoint computation. □

# Data complexity (bis)

**Theorem**

*Evaluating FO+$\mu^+$ is PTIME-complete in data complexity.*
*Evaluating FO+$\mu$ is PSPACE-complete in data complexity.*

**Proof.**

Hardness comes from descriptive complexity on ordered structures. □

# Plan

# Query optimization

- Goal: Given a query $q$ in some query language $\mathcal{Q}$ and a database $D$, find a query equivalent to $q$ on $D$ and faster to evaluate on $D$
- Here: $\mathcal{Q}$ in the relational calculus (or a fragment thereof), and one looks for a query faster on whatever database (we do not look $D$, we perform static analysis)
- In actual RDBMSs: $\mathcal{Q}$ is the set of query execution plans (a specialization of the relational algebra where implementations are chosen for each operator) and statistics on $D$ are used

# Global optimization

- We consider global optimization techniques, considering a query in its entirety (techniques on execution plans are more local, e.g., local rewritings)
- We formally define:

  Equivalence: $q \equiv q'$ if for all database $D$, $q(D) = q'(D)$
  Minimality: $q'$ is the "best" query equivalent to $q$ in $\mathcal{Q}$

# Containment and equivalence

## Definition

A query $q$ is contained in a query $q'$ (denoted $q \sqsubseteq q'$) if for all database $D$, $q(D) \subseteq q'(D)$

# Containment and equivalence

## Definition

A query $q$ is contained in a query $q'$ (denoted $q \sqsubseteq q'$) if for all database $D$, $q(D) \subseteq q'(D)$

## Proposition

$q \equiv q'$ iff $q \sqsubseteq q'$ and $q' \sqsubseteq q$.

## Proof.

Immediate. □

# Plan

# Case of conjunctive queries

- We consider conjunctive queries (CQ) of the form:

$$q(\mathbf{x}) \leftarrow \exists \mathbf{y} : R_1(\mathbf{z}_1) \wedge \cdots \wedge R_n(\mathbf{z}_n)$$

  where each $\mathbf{z}_i$ is a tuple of variables among $\mathbf{x}$ and $\mathbf{y}$, and where each $x_j$ appears at least in one $\mathbf{z}_j$
- Set semantics: for all database $D$, $q(D)$ is a finite set of tuples

# Homomorphism

## Definition

A homomorphism from a CQ $q$ to a CQ $q'$ is a function $\varphi$ from the variables $\mathbf{x}, \mathbf{y}$ of $q$ fo the variables $\mathbf{x}', \mathbf{y}'$ of $q'$ such that:

- $\varphi(\mathbf{x}) = \mathbf{x}'$
- for every atom $R(\mathbf{z}_i)$ of $q$, there exists an atom $R(\mathbf{z}'_{i'})$ of $q'$ such that $\varphi(\mathbf{z}_i) = \mathbf{z}'_{i'}$

## Definition

A homomorphism is an isomorphism if it is one-to-one and its converse is a homomorphism.

# Instance associated to a query

## Definition

For all conjunctive query

$$q(\mathbf{x}) \leftarrow \exists \mathbf{y} : R_1(\mathbf{z}_1) \wedge \cdots \wedge R_n(\mathbf{z}_n)$$

one can construct the instance associated to $q$, denoted $I_q$, where the active domain is $\{ a_z \mid z \in \mathbf{x} \cup \mathbf{y} \}$ and which is formed of the $n$ tuples $R(a_{z_{i1},\dots,z_{ik}})$ for $R(z_{i1}, \dots, z_{ik})$ atom of $q$

# Instance associated to a query

## Definition

For all conjunctive query

$$q(\mathbf{x}) \leftarrow \exists \mathbf{y} : R_1(\mathbf{z}_1) \wedge \cdots \wedge R_n(\mathbf{z}_n)$$

one can construct the instance associated to $q$, denoted $I_q$, where the active domain is $\{ a_z \mid z \in \mathbf{x} \cup \mathbf{y} \}$ and which is formed of the $n$ tuples $R(a_{z_{i1},\ldots,z_{ik}})$ for $R(z_{i1},\ldots,z_{ik})$ atom of $q$

## Proposition

For all CQs $q(\mathbf{x})$, $q'(\mathbf{x}')$, there exists a homomorphism from $q$ to $q'$ iff $(a_{x'_1},\ldots,a_{x'_j}) \in q(I_{q'})$.

# Homomorphism theorem

Theorem ([Chandra and Merlin, 1977])

*For all CQs $q$, $q'$, $q \sqsubseteq q'$ iff there exists a homomorphism from $q'$ to $q$.*

# Minimal query

## Definition

A conjunctive query is minimal if it has a minimal number of atoms among all equivalent conjunctive queries.

# Minimal query

## Definition

A conjunctive query is minimal if it has a minimal number of atoms among all equivalent conjunctive queries.

- Translation of a CQ to an algebra query: if there are $n$ atoms, we obtain $n-1$ joins
- Joins are the most costly operations of the relational algebra (bar cross products)
- Finding a minimal query amounts to global optimization

# Unicity of minimal query

## Proposition ([Chandra and Merlin, 1977])

Let $q$ be a CQ. Then there exists a CQ $q'$ obtained by removing atoms from $q$ which is minimal.

## Proof.

Consider a minimal query equivalent to $q$ and apply the homomorphism theorem. □

## Proposition ([Chandra and Merlin, 1977])

Let $q, q'$ be two equivalent minimal CQs. Then there exists an isomorphism from $q$ to $q'$.

## Proof.

Apply the homomorphism theorem. The image by the homomorphism is an equivalent minimal query. □

# Minimization algorithm

Apply the following procedure to minimize a query:

> *For every atom of the query, test if there exists an equivalent query not containing this atom, and thus if there exists a homomorphism sending this atom to another atom of the query. If so, delete it, and continue until obtaining an equivalent minimal query.*

# Complexity issues

## Proposition

The following problems are NP-complete:

- given two CQs $q$, $q'$, determine whether $q \sqsubseteq q'$
- given two CQs $q$, $q'$, determine whether $q \equiv q'$
- given a CQ $q$, determine if $q$ is non-minimal

## Proof.

NP-hardness is by reduction from 3-colorability, as for combined complexity of query evaluation. Membership in NP is direct. □

# Complexity issues

## Proposition

The following problems are NP-complete:

- given two CQs $q$, $q'$, determine whether $q \sqsubseteq q'$
- given two CQs $q$, $q'$, determine whether $q \equiv q'$
- given a CQ $q$, determine if $q$ is non-minimal

## Proof.

NP-hardness is by reduction from 3-colorability, as for combined complexity of query evaluation. Membership in NP is direct. □

NP-hard... in the queries. Queries may be small enough so that an exponential algorithm may not be an issue.

# Bag semantics [Chaudhuri and Vardi, 1993]

- In practice, RDBMSs implement a bag semantics
- Two queries in bag semantics are equivalent if and only if they are isomorphic (intuitively, because two similar but non isomorphic queries can introduce a different number of results)
- Query containment: $\Pi_2^P$-hard claimed (but not proved). Decidability (and precise complexity if decidable): open!

# Plan

# Satisfiability in the relational calculus

## Definition

A Boolean relational calcululs query $q$ is satisfiable if there exists a (finite) database $D$ such that $D \models q$.

# Satisfiability in the relational calculus

## Definition

A Boolean relational calcululs query $q$ is satisfiable if there exists a (finite) database $D$ such that $D \models q$.

## Theorem ([Trakhtenbrot, 1963])

*Satisfiability of the relational calculus (in the finite case) is undecidable.*

## Proof.

Reduction possible from the POST correspondence problem, technical, see [Abiteboul et al., 1995]. □

# Containment and equivalence of the calculus

### Theorem

*Containment and equivalence of relational calculus queries are undecidable and co-recursively enumerable.*

# Containment and equivalence of the calculus

## Theorem

*Containment and equivalence of relational calculus queries are undecidable and co-recursively enumerable.*

## Proof.

Undecidability is by direct reduction from the undecidability of satisfiability.
Co-recursive enumerability is shown directly, by enumerating possible counter-examples. □

# Plan

# Plan

# References

- 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]
- 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)

# 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


Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90, 1977. doi: 10.1145/800105.803397. URL http://doi.acm.org/10.1145/800105.803397.

Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 59–70, 1993. doi: 10.1145/153850.153856. URL http://doi.acm.org/10.1145/153850.153856.

# Bibliography III


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 IV


Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi: 10.1007/978-3-662-07003-1. URL http://dx.doi.org/10.1007/978-3-662-07003-1.

Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984. doi: 10.1137/0213035. URL http://dx.doi.org/10.1137/0213035.

Boris A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *American Mathematical Society Translations Series 2*, 23:1–5, 1963.

Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.