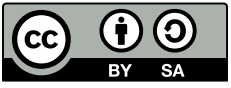
Introduction to Relational Database Management

Databases

Pierre Senellart



18 February 2026

Outline

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++, 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

Outline

Diversity of DBMSs

- Dozens of existing DBMSs that are broadly used (cf. `https://db-engines.com/en/ranking`)
- 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

In this course

- only mostly cover classical relational DBMSs (the most used, the most mature, by far!)
- Many aspects also relevant to other forms of DBMSs

Outline

Tables, schema, instance (informal)

- A relation is a table with columns (attributes) and rows (tuples)
- The schema tells us what columns exist (names) and what kind of values they contain (types)
- An instance is the current contents of the table (the set/bag of tuples stored right now)
- A database is a collection of named relations

We will later make this more precise.

Example database schema

- Database schema formed of 2 relation names, `Guest` and `Reservation`
- `Guest`: ((`id`, `INTEGER`), (`name`, `TEXT`), (`email`, `VARCHAR(256)`))
- `Reservation`: ((`id`, `INTEGER`), (`guest`, `INTEGER`), (`room`, `INTEGER`), (`arrival`, `DATE`), (`nights`, `INTEGER`))

Example database

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	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

Relational schemas: Formal definitions

We fix countably infinite sets (a universe):

- $\mathcal{L}$ 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, τ_i) with $L_i \in \mathcal{L}$, $\tau_i \in \mathcal{T}$ and such that all L_i are distinct

Definition

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

Universe in practice

- $\mathcal{L}$ 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), `VARCHAR` (character strings with maximum length), `DATE` (ISO 8601 representation of dates), etc.

Databases: Formal definitions

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.

Some notation

- If $A = (L, \tau)$ is the ith 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 ith 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 $t_1[\mathcal{A}] = t_2[\mathcal{A}]$; we mostly care about minimal keys
- 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 tuple t' of I^S with $t[\mathcal{A}] = t'[\mathcal{B}]$ (typically, $\mathcal{B}$ is a key of S and t' is unique)
- 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.

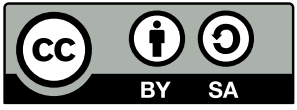
Variant: bag semantics

- 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 $\mathcal{V}$

Licensing



Bibliography I