TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

NoSQL Data Stores

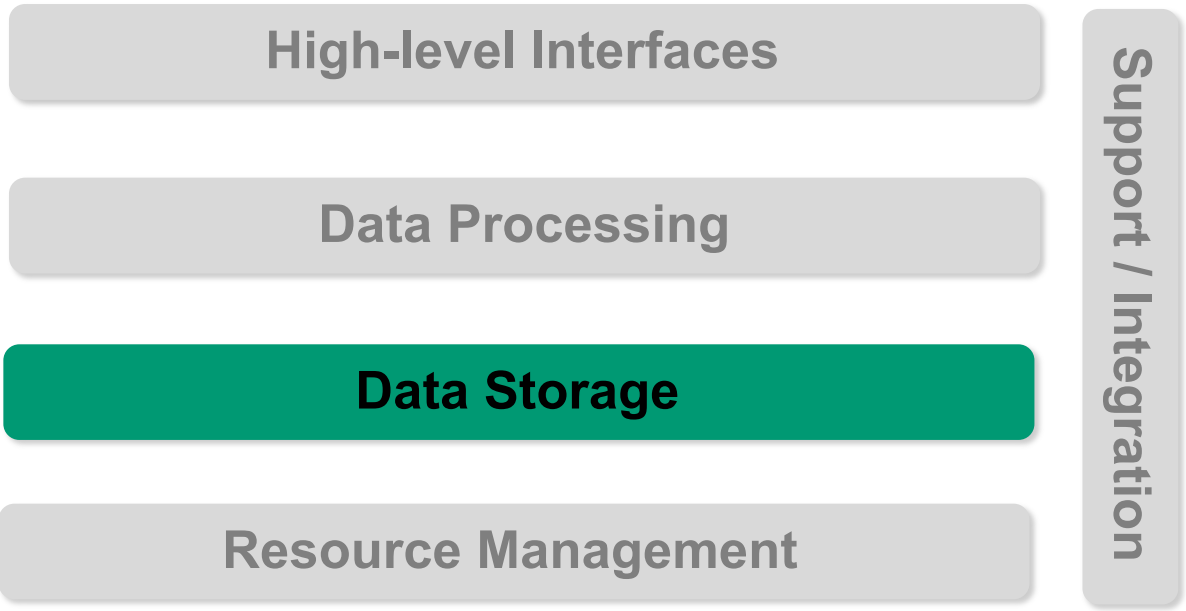
Corso di Sistemi e Architetture per Big Data

A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Traditional RDBMSs

- Relational DBMSs (RDBMSs)
  - Build on relational model
  - Traditional technology to store structured data in web and business applications
- SQL as standardized, powerful language
  - Rich language and toolset
  - Easy to use and integrate
  - Mature market (Oracle, MySQL, PostgreSQL, …)
- ACID guarantees

Recall: ACID

- **A**tomicity
  - All or nothing: transactions do not occur partially; failure leads to a full abort
- **C**onsistency
  - Rule-following: the database remains structurally sound and "correct" before and after a change
- **I**solation
  - Independence: uncommitted changes are hidden from other concurrent transactions
- **D**urability
  - Permanence: committed data survives system failures by being written to non-volatile disk

RDBMS constraints

- Domain constraints
  - Restrict domain or set of possible values for an attribute
    - E.g., `Age > 0`
- Entity integrity constraints
  - Every table must have a primary key and it cannot be null
- Referential integrity constraints
  - Maintain consistency among related tables: every foreign key value in a table must have a matching primary key in the other table
    - E.g, you cannot create an `Order` for a `Customer_ID` that does not exist in the `Customers` table
- Foreign key
  - To cross-reference between multiple relations: it is a key in a relation that matches the primary key of another relation

Pros and cons of RDBMS

Pros

- ✓ Well-defined consistency model
- ✓ ACID guarantees
- ✓ Relational integrity (entity and referential constraints)
- ✓ Well suited for OLTP applications
- ✓ Solid theoretical foundation
- ✓ Stable and standardized DBMS platforms
- ✓ Mature and widely understood

Cons

- ✗ Horizontal scaling is more difficult
- ✗ Less flexible for semi-structured or evolving data
- ✗ Queries require full knowledge of DB schema
- ✗ Commercial DBMSs are expensive (but open-source alternatives)
- ✗ Some DBMSs have field size limits
- ✗ Data integration from multiple RDBMSs can be cumbersome

RDBMS challenges

- Modern applications introduce new requirements
  - Workload spikes
  - Internet-scale data volumes
  - High read/write throughput
  - Frequent schema changes
- Traditional RDBMSs struggle with these
  - Not originally designed for distributed environments
  - Vertical scaling has limits
- How to scale RDBMSs?
  1. **Replication**
  2. **Partitioning** (or **sharding**)

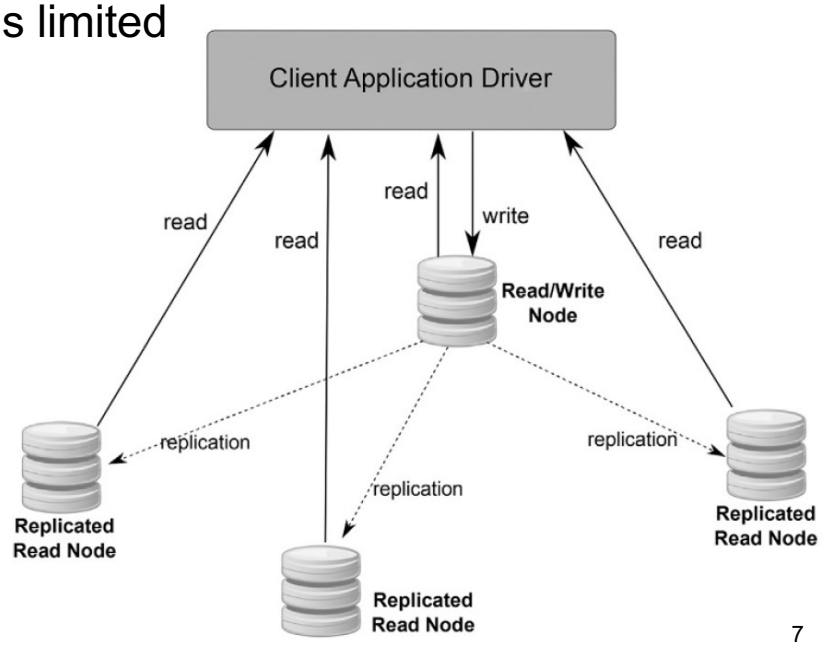
Replication

- Replicate data across servers
  - Common architecture: primary-backup with master/workers

✓ Replication improves availability and read scalability

✗ Write scalability is limited

- Alternatives
  - Multi-master
  - Partitioning



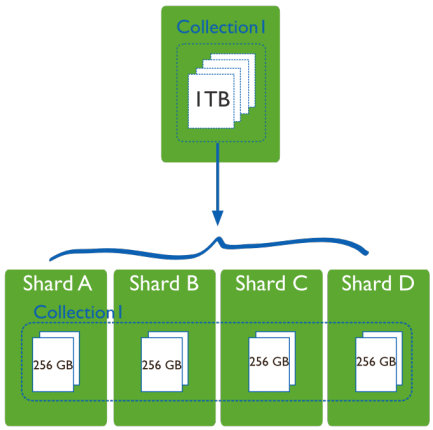
Partitioning (or sharding)

- Horizontal partitioning of data across multiple servers
  - Each server stores a subset of dataset (a shard)

✓ Read and write operations scale

✗ Transactions across multiple partitions are difficult and expensive, same for joins

- Shard assignment: consistent hashing can be use to determine *which* server stores a shard
  - Both data items and servers are hashed
  - Mapped into the same hash ID space
  - Hash determines which server is responsible for data



NoSQL data stores

- NoSQL = **Not Only SQL**
  - SQL-style querying is not the primary focus
- Designed to provide more flexibility and scalability than traditional RDBMSs

NoSQL data stores: characteristics

- Flexible data model
  - Support flexible schemas
  - No requirement for fixed rows or predefined table schema
  - Well suited for agile and evolving applications
- Horizontal scaling
  - Data and processing partitioned across multiple nodes
- High availability
  - Data replication across nodes, often geo-distributed
- Typically based on shared-nothing architecture
  - Some graph databases differ
- Often avoid complex operations such as joins
- Often support weaker consistency models
  - Follow **BASE** rather than ACID: trade-off between consistency, availability, and performance

ACID vs BASE: ACID

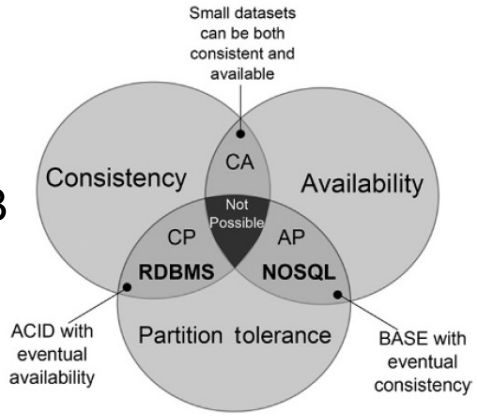
- Two design philosophies at opposite ends of the consistency-availability spectrum
  - Keep in mind **CAP theorem:** *pick two of **C**onsistency, **A**vailability and **P**artition tolerance*
- ACID: traditional approach for RDBMSs
  - Pessimistic concurrency control: prevents conflicts before they occur
    - Often implemented using write locks
    - Locking may cause performance degradation and deadlocks
  - Difficult to scale for large-scale distributed systems (latency!)

ACID vs BASE: BASE

- **BASE**: **B**asically **A**vailable, **S**oft state, **E**ventual consistency
  - **B**asically **A**vailable: system remains available most of the time, there could exist a subsystem temporarily unavailable
  - **S**oft state: data is not durable that is, its persistence is in the hand of the user that must take care of refreshing it
  - **E**ventually consistent: if no new updates occur, all replicas converge to the same state
- Optimistic approach
  - Let conflicts occur, detect and resolve
  - Conflict resolution
    - *Conditional updates*: test value before update
    - *Save both updates*: versioning/merge conflicting updates

NoSQL and consistency

- Key distinction from RDBMS
  - RDBMS: typically, strong consistency, classified as CA or CP systems (depending on configuration)
- Majority of NoSQL systems provide eventual consistency (classified as AP systems)
- Some NoSQL systems support strong consistency or *tunable* consistency
  - E.g., Cassandra and MongoDB



Pros and cons of NoSQL

Pros

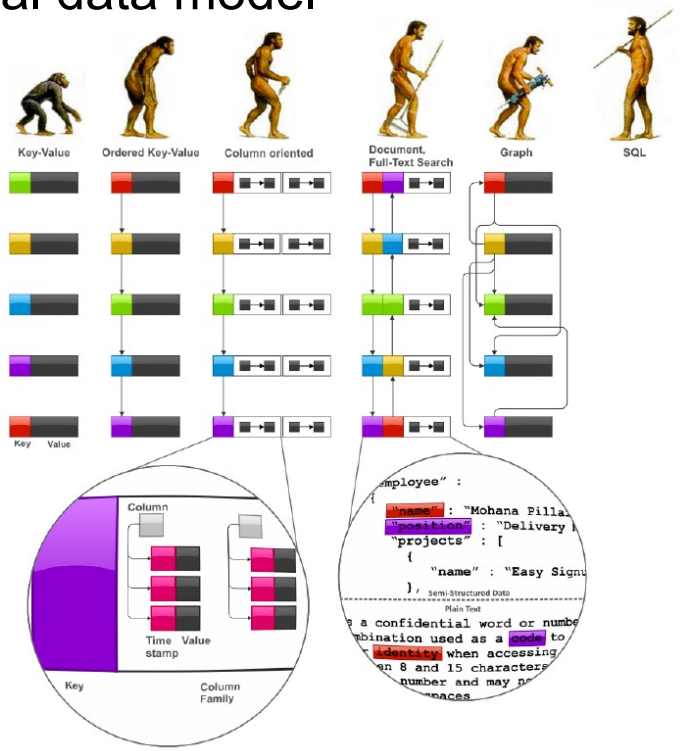
- ✓ Easy horizontal scaling
- ✓ High performance for massive datasets
- ✓ Data sharing across multiple servers
- ✓ High availability and fault tolerance through replication
- ✓ Many open-source options
- ✓ Flexible schema, support unstructured or semi-structured data
- ✓ Support for complex data structures and objects
- ✓ Fast data retrieval, suitable for real-time apps

Cons

- ✗ Many do not support ACID, less suitable for OLTP apps
- ✗ No standard data → no well-defined approach for design
- ✗ Lack of standardization (e.g., no standard query language)
- ✗ Many do not support joins
- ✗ Lack of reference model → can lead to vendor lock-in

NoSQL data models

- A number of largely diverse data stores, not based on relational data model



NoSQL data models

- *Data model*: set of constructs for representing information
  - Example: relational model → tables, rows, columns
- *Storage model*: how the data store management system stores and manipulates data internally
- Data model is usually independent of storage model
- NoSQL data models:
  - Aggregate-oriented models: **key-value (KV)**, **document**, and **column-family**
  - **Graph-based** models

Aggregate

- Data as single unit with a complex structure
  - More structured than just a set of tuples
  - E.g., records with nested fields, arrays, or sub-records
- Aggregate pattern in Domain-Driven Design
  - Cluster of domain objects treated as a single unit (e.g., order and its items, playlist and its songs)
  - Unit for data manipulation and consistency management

https://martinfowler.com/bliki/DDD_Aggregate.html

Aggregate

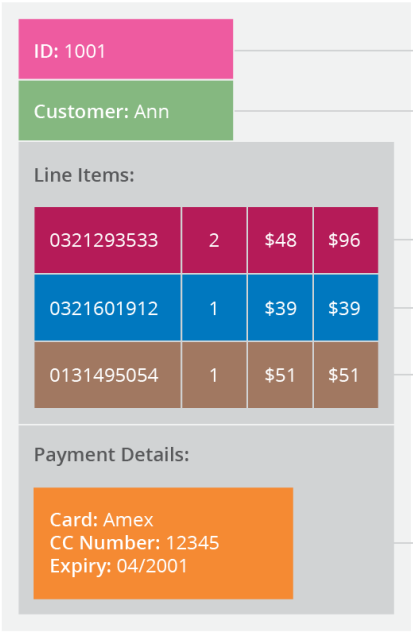
- Pros
  - Easier for application programmers
  - Simple for data stores to handle operations

- Trade-offs
  - Data redundancy: aggregation can lead to data duplication
  - Update complexity: updates to nested arrays or sub-objects require careful handling; more logic may be needed to maintain consistency

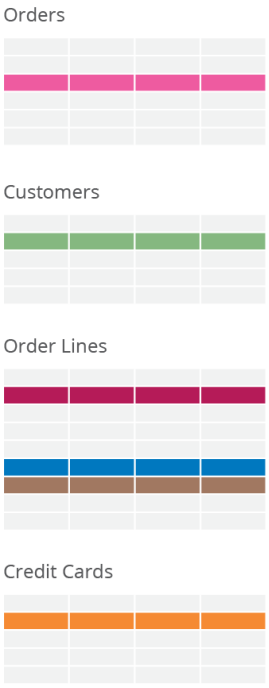
  https://www.thoughtworks.com/insights/blog/nosql-databases-overview

Aggregates: example

- With NoSQL



- With RDBMS



Transactions?

- Relational databases support ACID transactions
  - Can span multiple tables
- Aggregate-oriented data stores
  - Support atomic transactions only *within* a single aggregate
  - ACID transactions that span multiple aggregates are usually not supported
    - Update over multiple aggregates can result in inconsistent reads
  - ☞Design aggregates carefully to minimize cross-aggregate updates
- Graph databases often support ACID transactions