Systems And Methods For Big And Unstructured Data

Christian Rossi

Academic Year 2024-2025

Abstract

The course is structured around three main parts. The first part focuses on approaches to Big Data management, addressing various challenges and dimensions associated with it. Key topics include the data engineering and data science pipeline, enterprise-scale data management, and the trade-offs between scalability, persistency, and volatility. It also covers issues related to cross-source data integration, the implications of the CAP theorem, the evolution of transactional properties from ACID to BASE, as well as data sharding, replication, and cloud-based scalable data processing.

The second part delves into systems and models for handling Big and unstructured data. It examines different types of databases, such as graph, semantic, columnar, document-oriented, key-value, and IR-based databases. Each type is analyzed across five dimensions: data model, query languages, data distribution, non-functional aspects, and architectural solutions.

The final part explores methods for designing applications that utilize unstructured data. It covers modeling languages and methodologies within the data engineering pipeline, along with schema-less, implicit-schema, and schema-on-read approaches to application design.

Contents

CHAPTER 1

Introduction

1.1 Big data

Controlling the full potential of big data requires a well-structured data management process that spans every stage of the data pipeline. The essential components of this process include:

1. *Data collection*: the foundation of any big data initiative lies in gathering information from diverse sources.

2. *Data analysis*: the data must be carefully analyzed to extract meaningful patterns, trends, and insights. This step often involves tailoring the analysis to meet the needs of specific stakeholders. Techniques include:

    - Descriptive analysis: providing a clear picture of current trends and performance.
    - Predictive analysis: using historical data to forecast future scenarios and behaviors.

3. *Value creation*: the ultimate goal of the data pipeline is to transform raw data into actionable insights that drive value for organizations, enabling informed decision-making and innovation.

Big data's growing prominence is driven by several critical factors:

- *Decreasing storage costs*: advances in storage technology have significantly reduced costs, allowing organizations to store massive datasets affordably. This affordability makes it practical to collect and analyze large volumes of data on a regular basis.

- *Ubiquitous data generation*: in our digital age, data is generated everywhere.

- *Rapid data growth*: the volume of data is increasing at a pace far greater than IT budgets. This mismatch underscores the urgency for scalable and efficient data management solutions to handle the ever-growing data demands across industries.

Dimension	**Description**
Volume	Refers to the immense scale of data generated and stored. Big data encompasses vast quantities, ranging from terabytes to exabytes, made possible by increasingly affordable storage solutions
Variety	Describes the diverse forms in which data is available. Big data includes structured data, unstructured data, and multimedia content
Velocity	Represents the speed at which data is generated, processed, and analyzed. Big data often involves real-time or near-real-time data streams, enabling rapid decision-making within fractions of a second
Veracity	Concerns the uncertainty and reliability of data. Big data often includes information that may be imprecise, incomplete, or uncertain, requiring robust methods to manage and ensure data accuracy and predictability

Table 1.1: The four V's of Big Data

1.1.1 Data analysis

As the volume of data continues to grow, our methods for addressing data-related challenges must adapt accordingly. Traditional approaches to data analysis were constrained by processing limitations, often relying on small subsets of data. In contrast, big data enables a paradigm shift, allowing for the analysis of entire datasets, leading to more comprehensive understanding and deeper insights.

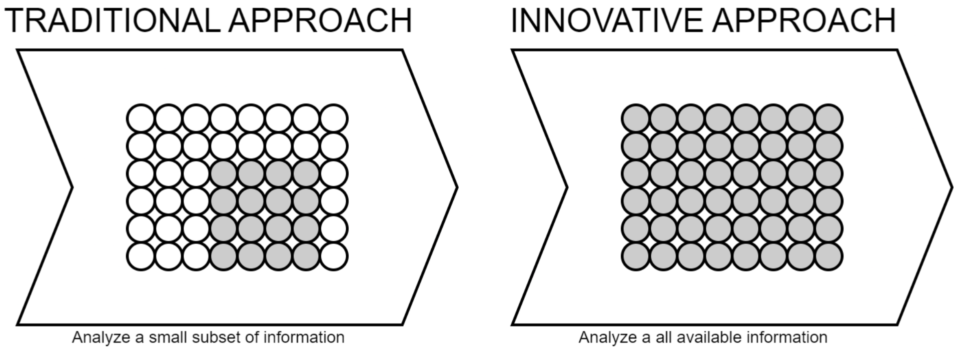

TRADITIONAL APPROACH
INNOVATIVE APPROACH
Analyze a small subset of information
Analyze a all available information

Figure 1.1: Data analysis

In traditional analysis, a hypothesis is formed and tested against a small, pre-selected dataset. While useful, this method limits discoveries to the scope of the sample. In the big data approach, all available data is explored, uncovering correlations and patterns without requiring predefined hypotheses. This data-driven exploration reveals insights that traditional methods might overlook.

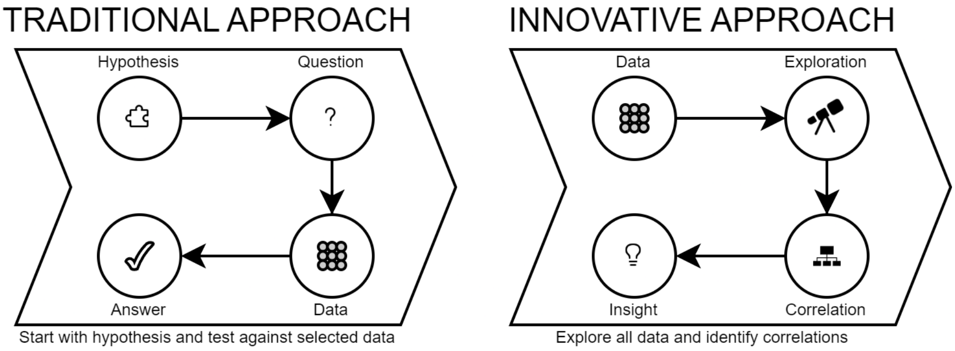


Figure 1.2: Data-driven analysis

Traditional methods require extensive data cleansing before analysis, resulting in a smaller but highly organized dataset. Big data, on the other hand, embraces the messiness of raw data, analyzing it first and cleansing as necessary. This approach enables the processing of larger volumes of less structured data.

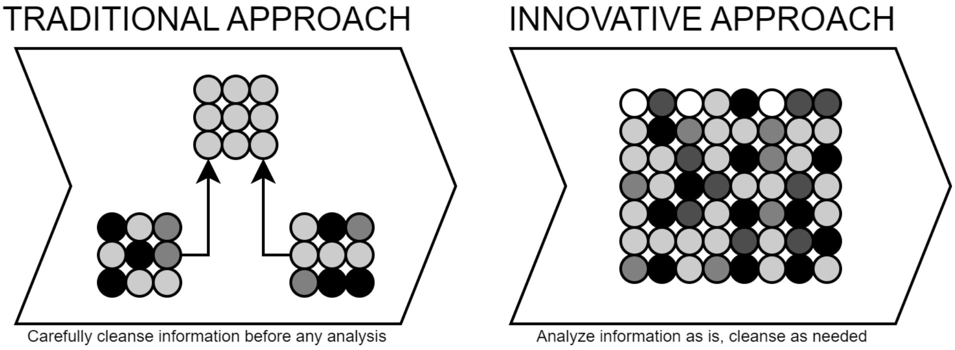

TRADITIONAL APPROACH
Carefully cleanse information before any analysis
INNOVATIVE APPROACH
Analyze information as is, cleanse as needed

Figure 1.3: Data cleaning

Traditional techniques often involve analyzing data only after it has been processed and stored in a data warehouse or mart. Big data leverages real-time analysis, processing data as it is generated. This enables timely insights and faster decision-making.

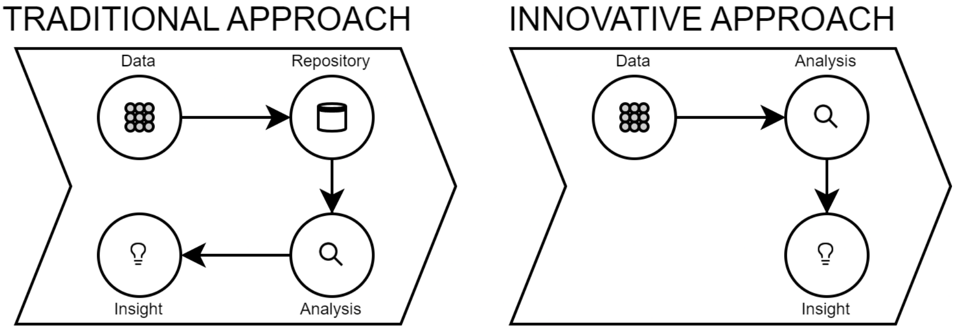


Figure 1.4: Real-time analysis

1.2 Relational database

Relational databases are structured systems designed to organize, store, and manage data efficiently. The design of a database typically occurs at three distinct levels:

- *Conceptual database design*: information model that is independent of physical implementation details, providing a high-level representation of the enterprise's data requirements.
- *Logical database design*: organizational database based on a specific data model ensuring logical structure and consistency.
- *Physical database design*: implementation of the database using specific data storage structures and access methods.

1.2.1 Characteristics

Entity In an entity-relationship database model, an entity represents a distinct, real-world object that can be uniquely identified. Entities are characterized by attributes, which describe their properties. Entities that share the same attributes are grouped into an entity set, and each set is distinguished by a primary key. The attributes of an entity belong to defined domains, which specify the range of valid values for each attribute.

Relationship Relationships between entities form another core aspect of relational databases. A relationship is an association between two or more entities, and all relationships of the same type are grouped into relationship sets. These relationships can include descriptive attributes, which provide additional context, and are defined by the participating entities. The concept of cardinality further specifies the number of connections that can exist between entities in a relationship, such as one-to-one or one-to-many associations.

Hierarchies and aggregations To enhance the flexibility of the model, relational databases often incorporate hierarchies and aggregations. ISA hierarchies allow for subclassing, where specific subclasses of an entity set can include unique attributes that further describe the entities within them. Aggregation offers a way to model more complex scenarios where a relationship itself needs to participate in another relationship. By treating a relationship set as though it were an entity set, it becomes possible to incorporate that relationship into higher-level associations.

1.2.2 Design

A critical aspect of database design is determining whether a concept should be modeled as an entity or a relationship. This decision significantly impacts the structure and clarity of the model.

The entity-relationship model is a powerful tool for capturing the semantics of data, offering a rich representation of relationships and constraints. However, not all constraints can be expressed within entity-relationship diagrams. Certain advanced constraints may need to be implemented in other layers.

1.2.3 Definitions

Relational databases revolutionized data management by formalizing the concept of structured data organization. SQL, based on this relational model, became commercially available in Database Management Systems in 1981.

The relational model relies on the mathematical notion of a relation, which is naturally represented as a table. Each relation is defined by a Cartesian product, which forms the foundation of the relational structure.

Definition (*Cartesian product*)**.** Given n sets $D_1, D_2, \dots, D_n$, the Cartesian product, denoted as $D_1 \times D_2 \times \cdots \times D_n$, is the set of all ordered n-tuples $(d_1, d_2, \dots, d_n)$ such that $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$.

Definition (*Mathematical relation*)**.** A mathematical relation on $D_1, D_2, \dots, D_n$ is a subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$.

Definition (*Relation*)**.** A relation is a set of ordered n-tuples $(d_1, d_2, \dots, d_n)$, where $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$.

Definition (*Relation domain*)**.** The sets $D_1, D_2, \dots, D_n$ are referred to as the domains of the relation.

Definition (*Relation degree*)**.** The number n is called the degree of the relation.

Definition (*Relation cardinality*)**.** The number of n-tuples in a relation is termed its cardinality.

1.2.3.1 Schema and instances

Definition (*Relation schema*)**.** A relation schema consists of a relation's name R and a set of attributes $A_1, \dots, A_n$, and it is denoted as $R(A_1, \dots, A_n)$.

Definition (*Database schema*)**.** A database schema is a collection of relation schemas with unique names, denoted as $R = \{R_1(X_1), \dots, R_n(X_n)\}$.

Definition (*Relation's instance*)**.** A relation instance on a schema $R(X)$ is a set r of tuples on X.

Definition (*Database's instance*)**.** A database instance for a schema $R = \{R_1(X_1), \dots\}$ is a set of relations $r = \{r_1, \dots\}$.

The relational model enforces a strict structure on data to ensure consistency and logical organization. In this model, all information is represented as tuples, which must conform to predefined relation schemas. This rigid framework provides clarity but also imposes certain limitations on how data is modeled and stored.

Null values Null values are used to represent missing or inapplicable information in relational databases. These nulls can be categorized into three distinct types. Despite these distinctions, most database management systems do not differentiate between these types of nulls. Instead, they implicitly treat all nulls as"no-information values.

Integrity constraint An integrity constraint is a condition that must hold true for all valid database instances. These constraints act as predicates, ensuring that a database instance is legal only if it satisfies all specified conditions.

1.2.3.2 Keys

A key is a fundamental concept in the relational model, representing a set of attributes that uniquely identifies tuples in a relation.

Definition (*Superkey*)**.** A set of attributes K is a superkey for a relation r if no two distinct tuples t_1 and t_2 in r share the same values for K.

Definition (*Key*)**.** The set of attributes K is a key for r if it is a minimal superkey, meaning no proper subset of K is itself a superkey.

The keys can be classified as:

- *Primary key*: to maintain data integrity, null values are not permitted in primary keys. Each relation must have a designated primary key, which uniquely identifies its tuples. In database schema notation, primary key attributes are underlined for clarity. Primary keys also play a critical role in establishing references between relations. Through these references, data consistency is maintained across the database.

- *Foreign key*: enable relationships between tuples in different relations. A foreign key in one relation references the primary key in another, establishing a link between the two. To ensure consistency, referential integrity constraints are imposed. These constraints guarantee that every foreign key value corresponds to an existing primary key value in the referenced relation. Violating these constraints would result in an inconsistent database state.

1.2.4 Schema transformation

Transforming an entity-relationship diagram into a relational database schema involves a systematic approach to map entities and relationships into relational tables. The process consists of the following key steps:

1. *Mapping entities to tables*: each entity in the ER diagram is represented as a separate table in the relational schema. The attributes of the entity become the columns of the table, while each instance of the entity set corresponds to a row in the table.

2. *Handle relationships*: relationships between entities are incorporated into the schema based on their cardinality and complexity.

1.3 Data architecture

A well-defined data schema ensures typing, coherence, and uniformity within a system, serving as the foundation for reliable data management. Central to this are transactions, which provide the mechanism to perform atomic and consistent operations on the database.

Definition (*Transaction*)**.** A transaction is the smallest unit of work executed by an application within a database system.

A transaction ensures that operations are executed in a controlled and reliable manner. Transactions are demarcated by two key commands:

- `BEGIN TRANSACTION`: marks the start of the transaction.

- `END TRANSACTION`: concludes the transaction, after which one of the following outcomes occurs:
  - `COMMIT`: finalizes the transaction, making all changes permanent.
  - `ROLLBACK`: aborts the transaction, reverting the system to the state it was in before the transaction began.

Online Transaction Processing systems play a crucial role in managing and executing transactions for multiple, concurrent applications.

Definition (*Online Transaction Processing*)**.** An Online Transaction Processing system is a platform that defines and processes transactions on behalf of various applications running simultaneously.

1.3.1 Partitioning

Data partitioning is an essential technique for achieving scalability and efficient distribution of data in large-scale database systems. By dividing the data into smaller chunks, the system can distribute the load across multiple storage nodes, thus enhancing performance and enabling horizontal scaling. The primary goal of partitioning is to optimize data retrieval and improve the overall efficiency of database operations.

There are two primary methods for partitioning data:

- *Horizontal partitioning* (sharding): different rows are stored across separate nodes. This is particularly useful in large-scale systems, where distributing data across multiple machines helps spread the load, enabling the system to scale out horizontally.
- *Vertical partitioning*: different columns of a table are stored on different nodes. This method is beneficial when certain columns are accessed more frequently than others, allowing for better optimization of data retrieval.

By distributing the data, both read and write operations can be performed more quickly, as each node handles a smaller subset of the total data. Additionally, partitioning reduces memory overhead, as it ensures that only relevant portions of the data are loaded into memory at any given time. Partitioning also facilitates scalability, enabling a system to grow by adding more nodes to handle increasing data volume or traffic.

One of the major risks is potential data loss or inconsistency, particularly if partitions are not managed properly. Node failures or mismanagement of partitions can lead to problems, especially when ensuring that the data remains consistent across different nodes. Moreover, managing partitions adds complexity to the database system, requiring sophisticated strategies to maintain data integrity and fault tolerance.

1.3.2 Replication

Data replication is a key strategy for ensuring fault-tolerance and reliability within distributed database systems. The core objective of replication is to maintain multiple copies of the database across various nodes in the system. This redundancy provides robust protection against data loss, as the failure of one node or even multiple nodes does not result in the loss of the data, since other copies are available.

One of the main advantages of replication is that it enhances data read performance. Since multiple copies of the data are distributed across different nodes, read requests can be served from the nearest or least-loaded node, leading to faster response times. Additionally, replication increases the overall reliability of the system, as the risk of losing all copies of the data is greatly diminished.

Replication introduces higher network overhead, as nodes must continuously synchronize with one another to ensure that all copies remain consistent. Furthermore, replication increases memory overhead because each node in the system stores a full duplicate of the dataset.

1.3.3 Scalability

Scalability is a fundamental goal when designing systems to handle varying loads. In modern systems, scalability often ties directly to the concept of elasticity.

Definition (*Elasticity*)**.** Elasticity refers to a system's ability to automatically adjust resources based on demand, either scaling up to accommodate higher loads or scaling down when demand decreases.

This capability ensures that the system operates efficiently without wasting resources or compromising performance. Elastic systems are essential in environments where workloads fluctuate, as they allow for the dynamic allocation of resources to maintain consistent performance while optimizing costs.

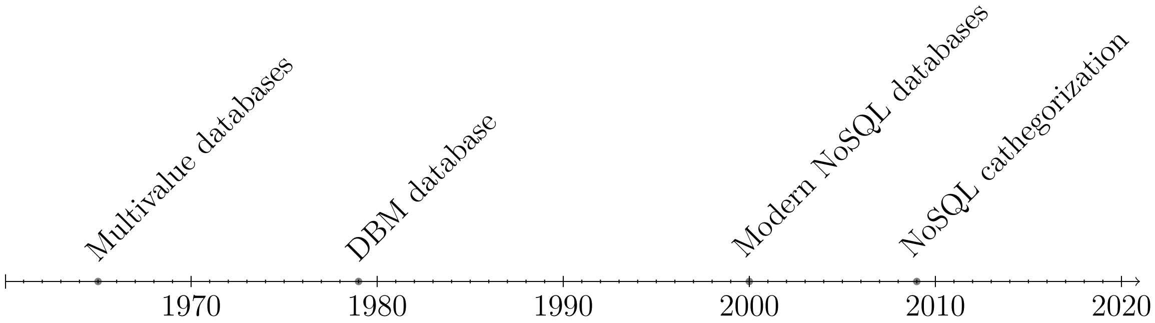
1.3.4 Data ingestion

Data ingestion is the process by which data is collected, imported, transferred, and loaded into a system for storage and future analysis. It involves extracting data from various sources, and then loading it into a storage system, often after making adjustments to ensure it is in a format that enhances storage efficiency and accessibility. This process may require transforming data to meet the system's specific requirements, which can include normalization, encoding, or formatting changes to ensure consistency across the dataset.

Wrangling Data wrangling is a critical step in the data ingestion pipeline that focuses on cleaning and transforming raw, unstructured data into a structured and usable format. This process involves several stages, including identifying and correcting errors, handling missing values, filtering out irrelevant information, and converting data into a format that is ready for analysis. The goal of data wrangling is to prepare the data for analysis by ensuring its quality, consistency, and structure.

1.4 NoSQL databases

NoSQL databases emerged as an alternative to traditional relational databases, offering greater flexibility and scalability.



NoSQL databases are designed to manage dynamic and unstructured data, often without requiring an explicit schema. This flexibility allows NoSQL databases to efficiently handle large-scale, constantly changing datasets.

The rise of Big Data has influenced a major shift in database design. NoSQL databases adopt a schema on read approach, meaning that data can be ingested without a predefined structure. The schema is only applied when the data is read or queried, offering more adaptability and scalability compared to traditional systems.

Object-relational mapping In traditional databases, object-relational mapping is used to bridge the gap between object-oriented programming languages and relational databases. Object-relational mapping helps overcome the impedance mismatch that arises when translating objects in code to rows in a relational database. NoSQL databases mitigate or even eliminate this issue by storing data in formats that align more naturally with the objects in application code.

Data lake NoSQL databases are often at the core of data lakes, which serve as large, centralized repositories for raw, unstructured, and structured data. Data lakes are designed to store data in its native format, allowing for future analysis without requiring immediate transformation into a rigid schema. This approach makes NoSQL databases highly compatible with data lakes, as they can efficiently store and manage vast amounts of diverse data that can be used for analysis at a later time.

Scalability One of the key strengths of NoSQL databases is their ability to scale horizontally. This approach is particularly effective for handling the massive datasets and high-throughput demands often found in Big Data applications. By scaling out across multiple machines, NoSQL databases can meet the increasing demands of modern applications without sacrificing performance or reliability.

1.4.1 CAP theorem

The CAP theorem describes the inherent trade-offs faced by distributed systems.

Theorem 1.4.1. *A distributed system cannot simultaneously guarantee all three of the following properties:*

- *Consistency: every node in the system has the same view of data at the same time.*
- *Availability: every request receives a response, regardless of success or failure.*
- *Partition tolerance: the system remains operational even if communication between nodes is disrupted due to network failures.*

In practice, distributed systems must make compromises. NoSQL databases often trade either consistency or availability depending on their specific use case. Based on the CAP theorem, systems are typically classified as follows:

- *CP*: these systems ensure data accuracy and integrity but may sacrifice availability during network failures.
- *AP*: these systems prioritize availability, allowing responses even if the data might be stale or inconsistent during partition events.

Choosing the right balance of these properties is essential for designing systems that align with performance, reliability, and scalability goals.

BASE properties Traditional databases adhere to the ACID principles (atomicity, consistency, isolation, durability) to ensure strict data reliability. In contrast, many NoSQL databases follow the BASE model, which offers a more flexible approach to consistency:

- *Basically Available*: the system prioritizes availability, even if it temporarily sacrifices consistency.
- *Soft state*: the system's state may evolve over time without new input, reflecting eventual changes.
- *Eventual consistency*: the system guarantees consistency over time, but intermediate states may remain inconsistent.

The BASE model is particularly suited to scenarios where high availability and scalability are critical. It enables systems to handle network partitions and high traffic volumes effectively while still converging to a consistent state.

1.4.2 NoSQL taxonomy

NoSQL databases can be classified into several types, each designed to address different data management needs, scalability, and performance requirements:

- *Graph databases*: these databases model data as nodes and relationships (edges), which is optimal for scenarios that involve complex relationships.
- *Documental databases*: data is stored as documents, typically in flexible formats like JSON, making them suitable for semi-structured or unstructured data. This type is ideal for applications requiring rich data models.
- *Key-value databases*: data is stored as simple key-value pairs, making these databases highly efficient for lookups based on keys.
- *Column stores*: data is stored in columns rather than rows, which is particularly useful for analytical queries and big data applications. Column stores excel at reading and processing large datasets quickly.

CHAPTER 2

NoSQL databases

2.1 Graph databases

Relational databases often struggle with efficiently managing and querying complex relationships between data entities. In contrast, graph databases are specifically designed to handle such tasks using graph structures, which consist of nodes (entities), edges (relationships), and properties. Graph databases have index-free adjacency, which means that each node directly references its adjacent nodes.

They not only connect nodes to other nodes but can also link nodes to properties, making the data structure highly flexible for relationship-focused queries. Graph databases are ideal fit for scenarios where relationships are central to the analysis:

- *High performance on relationship queries*: graph databases are optimized for associative datasets, such as social networks.
- *Natural fit for object-oriented models*: they inherently support hierarchical structures like parent-child relationships and object classification.
- *Efficient traversal*: because nodes directly point to adjacent nodes, queries that involve traversing relationships are much faster compared to relational databases.

Advantages	Easy to extend Easy to change
Disadvantages	Complexity growing with the number of elements Difficult query optimization

Pattern matching Pattern matching is a technique used to find specific structures or relationships within a graph by querying based on patterns of nodes and edges. This approach allows users to search for complex data relationships by specifying the nodes, types of relationships, and desired properties they want to match. It is powerful because it enables querying highly interconnected data quickly.

2.1.1 Neo4j

Neo4j, developed by Neo Technologies, is one of the leading and most popular graph databases available today. It is implemented in Java and is open-source, providing a robust platform for managing and querying graph data.

Feature	**Description**
Schema-free	Flexible data model that does not require a predefined schema
ACID compliant	Ensures atomicity, consistency, isolation, and durability
User-friendly	Easy to learn, set up, and use, even for new developers
Extensive documentation	Supported by a large, active developer community
Multi-language support	Compatible with Java, Python, Perl, Scala, and Cypher

Neo4j is primarily designed as an operational database rather than a dedicated analytics platform. It excels at managing relationships and provides efficient access to nodes and connections. However, it may be less suited for large-scale, full-graph analyses compared to specialized analytics engines.

2.1.1.1 Architecture

Neo4j's architecture consists of three primary layers:

- *Memory Layer*: stores records of nodes, relationships, types, and properties. Nodes and edges are managed separately, streamlining queries that target only specific elements. Neo4j attempts to load as much of the graph into RAM as possible to enable fast data access and analysis.
- *Operating System layer*: provides a cache to map elements in RAM to their counterparts in secondary storage, facilitating efficient memory management.
- *Execution environment*: as Neo4j is written in Java, it runs on the Java Virtual Machine, which also hosts APIs that allow users to interact with the database.

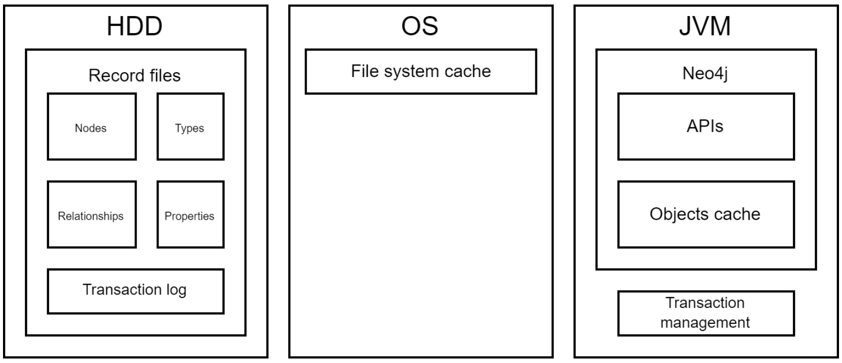


Figure 2.1: Neo4j architecture

Neo4j uses a declarative query language, Cypher. Each query is translated into an execution plan by the query optimizer, which then sends it to the query engine to execute and return results. For repeated queries with varying parameters, it's recommended to use parameterized queries to avoid redundant optimization for each execution.

2.1.1.2 Data Model

The Neo4j data model is based on three primary components:

- *Nodes*: represent entities, each labeled by types and equipped with attributes (properties).
- *Relationships*: define connections between nodes, providing context and capturing relationships between entities.
- *Indexes*: improve query performance by allowing fast lookups of nodes and relationships based on properties.

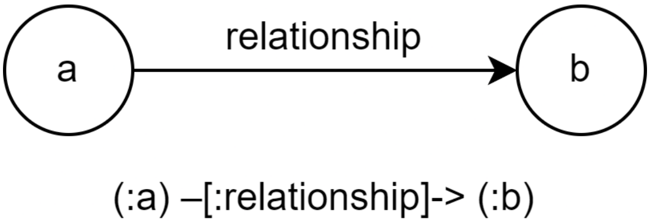


Figure 2.2: Neo4j data model

2.1.1.3 Query language

Cypher is the dedicated query language for Neo4j, designed to be both user-friendly and powerful. Its declarative nature allows users to specify what data they want to retrieve without needing to define how to obtain it, making query formulation straightforward.

Data creation and deletion Witch Cypher we can create a new node in the following way:

```
CREATE (node:Label {property: value, ... })
```

In the same way, we can create relationships between existing nodes:

```
CREATE (n1)-[r:RelationshipType {property: value, ...}]->(n2)
CREATE (n1)-[r:RelationshipType {property: value, ...}]-(n2)
```

Remeber that each node may have multiple labels to specify for example a group and a subgroup. We may also delete some nodes with all the relationships:

```
MATCH (node:Label {property: value, ... })
DETACH DELETE node
```

In this query the `delete` clause allows the removal of nodes and relationships, while the `detach` removes all the relationships before removing the nodes. This can also be applied to single relationships:

```
MATCH (n1)-[r:RelationshipType {property: value, ...}]->(n2)
DELETE r
```

Data importing Neo4j allows importing an entire graph from a csv file using the Cypher query language:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:data.csv" AS row
CREATE (:Label {property: row.Column1, ... })
```

The periodic commit is essential for handling large datasets efficiently, as it commits data in batches to ensure ACID compliance, preventing memory overload and maintaining transaction integrity.

Data merging To avoid creating duplicate nodes or relationships, use the merge operation:

```
MERGE (n:Label {property: ’value’})
ON CREATE
    SET n.property1 = ’new_value’
ON MATCH
    SET n.lastUpdated = date()
```

The merge clause checks if a node with the specified properties exists; if not, it creates it. The clause `on create` is used to set properties when the node is newly created, and `on match` allows updating properties if the node already exists. This approach can also be applied to relationships, ensuring no duplicate edges.

Indices and constraints To improve query performance, create an index on a specific property of a node label:

```
CREATE INDEX ON :Label(property)
```

This command creates an index on the specified property for nodes with the given label, speeding up searches on that property. To enforce data integrity, create a constraint on a specific property:

```
CREATE CONSTRAINT ON (n:Label)
ASSERT n.property IS UNIQUE
```

This command enforces uniqueness on the specified property for nodes with the given label, ensuring no duplicate values for that property across nodes.

Data querying The general structure of a Cypher query is as follows:

```
MATCH (n1)-[:RelationshipType]-(n2)
WITH n1, count(n2) AS relationCount
ORDER BY relationCount DESC
SKIP 1
LIMIT 3
RETURN DISTINCT n1
```

In this query:

- Aggregation functions like `count` can be utilized to calculate values.
- The `with` clause explicitly separates parts of the query and declares variables for subsequent sections.
- `skip` is used to bypass a specified number of results, while `limit` restricts the total number of results returned.
- `distinct` is used to return all different elements.

Additionally, appending an asterisk (*) to a relationship allows for retrieving all nodes that are not directly connected by that relationship type.

Cypher pattern	**Description**
`(n:Person)`	Node with the `Person` label.
`(n:Person:Swedish)`	Node with both `Person` and `Swedish` labels.
`(n:Person{name:$value})`	Node with the given properties.
`()-[r{name:$value}]-()`	Matches relationships with the given properties.
`(n)-->(m)`	Relationship from `n` to `m`.
`(n)--(m)`	Relationship in any direction between `n` and `m`.
`(n:Person)-->(m)`	Node `n` labeled `Person` with a relationship to `m`.
`(m)<-[:Know]-(n)`	`Know` relationship from `n` to `m`.
`(n)-[:Know|:Love]->(m)`	`Know` or `Love` relationship from `n` to `m`.
`(n)-[r]->(m)`	Binds relationship to `r`.
`(n)-[*1..5]->(m)`	Variable length path (1 to 5) from `n` to `m`.
`(n)-[*]->(m)`	Variable length path from `n` to `m`.
`(n)-[:Know]->(m{property:$value})`	`Know` relationship from `n` to `m` with a property.

Finding paths In Cypher, there are several functions available to identify paths within the graph between nodes. These functions allow you to efficiently navigate relationships and retrieve relevant data. To find the shortest paths between nodes, you can use the following Cypher queries:

- *Single shortest path*: this function retrieves a single shortest path between two nodes and the path can traverse up to six relationships:

  ```
  shortestPath((n1:Label)-[*..6]-(n2:Label))
  ```

- *All shortest paths*: if you want to find all possible shortest paths between two nodes, use this query. It ensures that every shortest path is considered:

  ```
  allShortestPaths((n1:Label)-[*..6]->(n2:Label))
  ```

To count the number of paths that match a specific pattern, you can use the following query. This example counts paths with a given structure originating from node `n` and extending through two relationships:

```
size((n)-->()-->())
```

2.2 Documental databases

In traditional relational databases, data is typically distributed across multiple tables, necessitating complex joins to retrieve related information. While this model has its advantages, it can become cumbersome for certain business applications that require a more cohesive and intuitive representation of data.

Document-oriented databases address this challenge by structuring data into self-contained documents. Each document encapsulates all relevant information, often combining what would traditionally be spread across multiple tables into a single, unified structure. This approach not only simplifies queries by eliminating the need for intricate joins but also enhances performance in scenarios with high read or write demands.

One of the key advantages of document-oriented databases is their inherent flexibility. They allow for schema-less design, enabling developers to adapt and evolve the data structure with minimal friction. This adaptability is particularly valuable in agile development environments where requirements can change frequently.

Moreover, the document model aligns closely with object-oriented programming paradigms. By mapping data structures directly to objects in code, it effectively eliminates the impedance mismatch often encountered when trying to bridge the gap between object-oriented designs and relational database schemas. As a result, development becomes more streamlined, and applications can handle data more naturally.

2.2.1 MongoDB

MongoDB is a highly popular, open-source, document-oriented database that offers flexibility, scalability, and performance for modern application development. Unlike traditional relational databases, MongoDB stores data in JSON-like documents, making it more dynamic and developer-friendly. Its schema-less design enables the effortless adaptation of data models as requirements evolve. Moreover, MongoDB supports automatic data sharding, which ensures seamless horizontal scaling across multiple servers.

Feature	**Description**
General-purpose design	Rich data model, advanced indexing, and powerful query language suited for diverse use cases like CMS and real-time analytics
Ease of use	Document model maps easily to object-oriented programming, with native drivers and simple setup for developers
Performance and scalability	In-memory operations and auto-sharding ensure high performance and seamless scaling without downtime
Security	SSL encryption, fine-grained access controls, and role-based authorization for robust data protection

2.2.1.1 Architecture

MongoDB relies on three key processes to manage and operate its architecture:

- *Mongod*: the primary process responsible for running the MongoDB database instance. It handles all core database operations, including data storage and query execution.
- *Mongos*: acts as the query router in a sharded cluster, distributing queries based on the sharding configuration. Multiple mongos instances can be deployed to improve performance and reduce network latency.
- *Mongo*: an interactive command-line shell that enables users to execute database commands and queries.

Indexes Indexes in MongoDB significantly enhance query performance by providing faster access to data. Key characteristics include:

- A default index is created on the id field (the primary key) of every document.
- Users can define additional indexes, including single-field and compound indexes, to optimize query execution or enforce constraints such as uniqueness.
- Array fields are supported, where separate index entries are created for each array element.
- Sparse indexes include entries only for documents that contain the indexed field, effectively ignoring documents without the field.
- Unique sparse indexes reject duplicate values but allow documents without the indexed field.

Sharding Sharding is MongoDB's strategy for partitioning large datasets across multiple servers, ensuring horizontal scaling and optimized performance. Sharding provides several benefits:

- *Scale*: efficiently handles massive workloads and ensures scalability as data volume grows.
- *Geo-locality*: supports geographically distributed deployments to enhance user experience across regions.
- *Hardware optimization*: allows intelligent data distribution across resources, balancing performance and cost.
- *Recovery time optimization*: reduces downtime during failures, supporting strict Recovery Time Objectives (RTO).

A shard key is defined by the data modeler and determines how MongoDB partitions data across shards. The sharding process involves defining a shard key, which determines how data is distributed across shards. The main steps are:

1. MongoDB begins with a single chunk of data. As the dataset grows, it automatically splits and migrates chunks to balance the load across shards.
2. Queries are routed directly to the relevant shard, reducing overhead.
3. Config servers store metadata about shard ranges and their locations. To ensure high availability, production systems typically use three config servers.

The sharding strategy can be:

- *Range-based*: data is partitioned by a continuous range of shard keys.
- *Hash-based*: MongoDB applies a hash to the shard key to distribute data randomly across shards, ensuring even distribution and minimizing hotspots.
- *Tag-aware*: specific shards are tagged to store particular subsets of data, such as region-based user data, optimizing geo-locality.

Usage	**Required strategy**
Scale	Range or hash
Geo-locality	Tag-aware
Hardware optimization	Tag-aware
Lower recovery times	Range or hash

MongoDB is designed to prioritize consistency and partition tolerance, making it an excellent choice for applications requiring scalability, performance, and flexibility.

2.2.1.2 Data model

MongoDB employs a flexible and intuitive data model based on JSON-like documents. This format is particularly well-suited for modern web and mobile applications due to its human readability, hierarchical structure, and adaptability to evolving requirements. Data is stored in contiguous regions, ensuring better data locality and faster access speeds.

MongoDB's design is optimized for contemporary application development, enabling developers to handle complex data relationships without the rigidity of traditional schemas. Large documents can be stored efficiently using GridFS, a feature that splits and distributes data across multiple files, supporting documents larger than the standard 16MB size limit.

Binary JSON MongoDB leverages BSON (Binary JSON), a binary-encoded serialization of JSON documents, to improve speed and efficiency. BSON enhances the traditional JSON format by introducing the following key features:

- *Extended data types*: BSON supports additional data types such as dates, byte arrays, and embedded objects, which are not natively available in JSON.
- *Optimized storage*: the binary format is more compact, reducing storage overhead and improving data transmission speeds.
- *Serialization efficiency*: BSON enables faster serialization and deserialization, enhancing database performance during read and write operations.

2.2.1.3 Query language

MongoDB provides a robust and flexible query language that supports all CRUD (Create, Read, Update, Delete) operations, enabling developers to interact with the database effectively. Its syntax is intuitive, leveraging JavaScript-like methods to simplify database management.

Create A new database is created as soon as a document is inserted into a collection. You can switch to a database (creating it implicitly) using:

```
use database_name
```

To explicitly create a collection with optional parameters like schema validation:

```
db.createCollection(name, options)
```

Documents can be inserted into a collection using:

```
db.<collection_name>.insert()
```

Indexes are data structures that store a small portion of the collection's data set in an easy to traverse form, ordered by the value of the field. Indexes support the efficient execution of some types of queries. Indexes are created with the `createIndex` operator which accepts a list of the fields with respect to which create the index and their corresponding ordering.

```
db.<collection_name>.createIndex()
```

Read Retrieve all documents in a collection with optional formatting for readability:

```
db.<collection_name>.find().pretty()
```

Filters can be added to the `find` function to narrow down results:

```
db.<collection_name>.find()
```

When collecting documents, it is possible to sort and limit the results. These operations can be performed through the `$sort` and `$limit` stages or using the `sort` and `limit` methods.

```
db.<collection_name>.find().sort().limit(number)
```

Filtering operations may exhibit different behaviors depending on the type of complex field a query accesses, such as subdocuments or arrays. Queries that evaluate one or more conditions on the fields of a subdocument are not subject to any particular behavioral change. However, queries that evaluate a single condition on the fields of documents within an array will return the main document if at least one of the documents in the array satisfies the condition. When multiple conditions are evaluated on the documents in an array field, they will be assessed individually for each document in the array. In this case, the main document is returned if, for each condition, there exists at least one document that satisfies it. It does not matter if only one document satisfies all conditions or if multiple documents each satisfy a single condition. If a query targets multiple conditions on the fields of the same document within an array, the `$elemMatch` stage must be used. The `$elemMatch` operator matches documents containing an array field with at least one element that satisfies all of the specified query criteria. When a collection consists of documents containing arrays, retrieving the content of those arrays may be useful. This can be accomplished using the `$unwind` stage. The `$unwind` stage reshapes the collection so that each document is replaced by a set of new documents, one for each element in the document's array. These new documents retain all fields from the original document and include a field with the name of the array field, which contains one of the elements.

MongoDB supports SQL-like aggregation for advanced data processing. Data passes through a pipeline where transformations and calculations are applied:

```
db.parts.aggregate()
```

Aggregate operations in MongoDB, aimed at grouping data with respect to one or more fields, are achieved using the `$group` stage within the `aggregate` method. This stage requires specifying the fields on which to perform the aggregation and the aggregation functions to apply. When the `$group` stage is applied, only the fields used for the aggregation or explicitly created within this stage will be available in subsequent stages of the pipeline. To perform a grouping operation on the entire dataset, a dummy `_id` can be used in the `$group` stage by setting `_id` to a constant value. This is why the `$` operator is crucial in the grouping stage for referencing specific fields. MongoDB's aggregation framework employs a pipeline model, where multiple stages are chained together to transform and analyze data. It is necessary to explicitly define all the stages in the pipeline. In addition to `$group`, the following are some commonly used stages:

- `$match`: filters documents based on specified conditions.
- `$project`: defines projections to reshape the document structure.
- `$unwind`: deconstructs array fields into separate documents.

- `$sort`: orders the documents based on specified fields.
- `$limit`: limits the number of documents passed to the next stage.

For complex data aggregation, MongoDB supports the map-reduce paradigm:

```
db.collection.mapReduce()
```

Update The general update command specifies selection criteria and new data:

```
db.<collection_name>.update(<select_criteria>,<updated_data>)
```

The `save` method replaces an existing document if it matches the identifier or inserts a new document if none exists:

```
db.<collection_name>.save()
```

Delete To delete an entire database:

```
db.dropDatabase()
```

To remove a collection:

```
db.<collection_name>.drop()
```

To delete documents based on a filter:

```
db.<collection_name>.remove(options)
```

Comparison operators MongoDB supports a wide range of comparison operators to filter query results efficiently. These operators enable precise matching and logical operations.

Name	**Description**
`$eq`	Matches values that are equal to a specified value
`$gt`, `$gte`	Matches values greater or equal to a specified value
`$lt`, `$lte`	Matches values less or equal to a specified value
`$ne`	Matches values that are not equal to a specified value
`$in`	Matches any of the values specified in an array
`$nin`	Matches none of the values specified in an array
`$or`	Joins query clauses with a logical OR

`$and`	Joins query clauses with a logical AND
`$not`	Inverts the effect of a query expression
`$nor`	Joins query clauses with a logical NOR
`$exists`	Matches documents that have a specified field
`$type`	Matches documents whose chosen field is of a specified type
`$text`	Matches documents based on text search on indexed fields
`$regex`	Matches documents based on a specified regular expression
`$where`	Matches documents based on a JavaScript expression

2.3 Key-value database

Key-value databases store data as a collection of key-value pairs. This structure allows for efficient retrieval of data. Conceptually, the key-value approach is analogous to indexing in relational databases, where a key serves as a reference to access the associated data object.

2.3.1 Redis

Redis supports atomic operations on its native data structures, ensuring that operations on a specific data type can be completed without interference from other operations. Redis can be used as a persistent database, a fast in-memory cache, or a message broker, making it a multi-purpose tool in modern architectures.

While Redis is not a direct replacement for relational databases or document stores, it complements them well. Best use cases for Redis are:

- Applications that require real-time data processing and fast access.
- Scenarios needing complex data structures, such as lists and sets, rather than basic key-value pairs.
- Situations where the dataset fits within memory, allowing for fast in-memory data retrieval.
- Non-critical datasets, as Redis persistence mechanisms can introduce some latency, which may be unsuitable for mission-critical applications.

The advantages of Redis are:

- *Performance*: Redis offers high-speed data access, ideal for real-time applications.

- *Availability*: replication and partitioning enhance data availability and fault tolerance.
- *Scalability*: Redis can be scaled to accommodate high-demand scenarios.
- *Portability*: Redis runs on most POSIX-compliant systems and has limited support for Windows.

2.3.1.1 Architecture

Redis, written in C, runs on most POSIX-compliant systems, with Linux recommended for production environments. Although Redis is single-threaded, it achieves scalability across multiple CPU cores by allowing multiple Redis instances to run in parallel. With constant-time complexity for many commands, Redis remains efficient even with high data volumes.

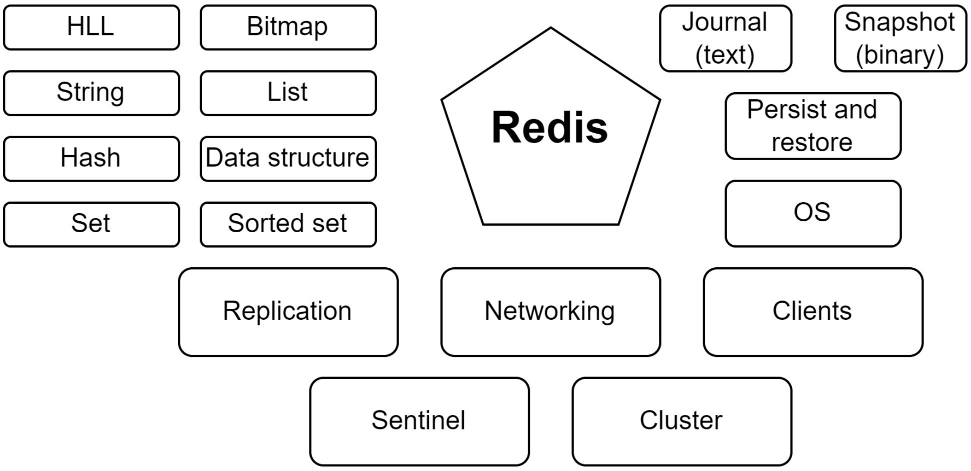


Figure 2.3: Redis architecture

Redis offers two persistence mechanisms:

- *Redis Database Snapshots*: captures a snapshot of the dataset at specified intervals.
- *Append-Only File*: logs every write operation, ensuring recovery by replaying commands if Redis restarts.

Redis enables master-slave replication, where one master Redis instance can synchronize with multiple read-only slave instances. Clients can read data from both master and slave nodes, but only write to the master by default. Redis also supports data partitioning across multiple hosts through:

- *Client-side partitioning*: client code manages data distribution.
- *Proxy-based partitioning*: uses a proxy layer to distribute requests.
- *Query router partitioning*: Redis Cluster automatically routes requests to the appropriate node.

Cofigurations Redis can be deployed in various configurations:

- *Standalone*: basic setup with optional master-slave replication for read offloading and redundancy. No automatic failover.
- *Sentinel*: provides automated failover in a master-slave topology, promoting a slave to master if the primary fails. Data is not distributed across nodes.
- *Twemproxy*: functions as a proxy to distribute data across standalone Redis instances, supporting consistent hashing and basic partitioning.
- *Cluster*: Redis Cluster distributes data across multiple instances with built-in failover and divides the keyspace into hash slots, where each node holds a subset of the hash slots.

2.3.1.2 Data model

The Redis data model is centered around key-value pairs, with additional data types for more complex storage needs.

Data type	Description
Strings	Basic key-value pairs, suitable for caching and counters
Lists	Ordered collections, useful for queues
Sets	Unordered unique collections, great for tags and unique items
Sorted sets	Sets with key, ideal for rankings
Hashes	Field-value pairs within a key, good for storing objects
Bitmaps	Bit-level data, useful for flags and tracking events
HyperLogLogs	Probabilistic unique counters with low memory usage
Streams	Log-like data for real-time processing and event sourcing

2.3.1.3 Query language

Redis uses a command-based language tailored to its data types. Commands are specific to the type of data being manipulated, ensuring efficient data access and manipulation for diverse data structures.

Strings The basic commands on strings are:

```
/* get and set strings */
SET string_key string_value
GET string_key
/* set or increment numbers values */
```

```
SET string_key 1
INCRBY string_key 1
/* get and set multiple keys at once */
MGET string_key string_key
MSET string_key string_value
/* get the length of a string */
STRLEN string_key
/* update a value retrieving the old one */
GETSET string_key string_value
```

Keys The basic commands on keys are:

```
/* key removal */
DEL key_value
/* test for existence */
EXISTS key_value
/* get the type of a key */
TYPE key_value
/* set an expiration time to a key */
EXPIRE key_value 10
/* get key time-to-live */
TTL key_value
```

List The basic commands on list are:

```
/* push on either end */
RPUSH key_value string
LPUSH key_value string
/* pop from either end */
RPOP key_value
LPOP key_value
/* blocking pop on either end */
BRPOP key_value
BLPOP key_value
/* pop and Push to another list */
RPOPLPUSH src_key_value dst_key_value
/* get an element by index on either end */
LINDEX key_value
/* get all list elements */
LRANGE key_value 0-1
```

Hash The basic commands on hash are:

```
/* set a hashed value */
HSET key:key_value field value
/* set multiple fields */
HMSET key:key_value lastfield Smith visits 1
/* get a hashed value */
HGET key:key_value field
```

```
/* get all the values in a hash */
HGETALL key:key_value
/* increment a hashed value */
HINCRBY key:key_value visits 1
```

Sets The basic commands on sets are:

```
/* add member to a set */
SADD key value
/* pop a random element */
SPOP key
/* get all elements */
SMEMBERS key
/* intersect multiple sets */
SINTER key key
/* union multiple sets */
SUNION key key
/* differentiate multiple sets */
SDIFF key key
```

Sorted sets The basic commands on sorted sets are:

```
/* add member to a sorted set */
ZADD key key_value value
/* get the rank of a member */
ZRANK key value
/* get elements by score range */
ZRANGEBYSCORE key 200 +inf WITHSCORES
/* increment score of member */
ZINCRBY key 10 value
/* remove range by score */
ZREMRANGEBYSCORE key 0 key_value
```

2.3.2 Memcached

Memcached is an open-source, distributed memory caching system created in 2003 by Brad Fitzpatrick to boost the performance of dynamic web applications by reducing database load. Using a key-value dictionary model, Memcached is particularly useful for storing frequently accessed, computationally expensive, or commonly shared data in memory, allowing applications to access it quickly. Originally intended to speed up dynamic websites like LiveJournal, Memcached is now widely used to cache data temporarily, ensuring faster response times without putting undue strain on databases.

Technically, Memcached operates as a server that clients can access over TCP or UDP, and multiple Memcached servers can be grouped into pools to expand available cache memory. This setup allows for a high degree of flexibility and scalability, particularly in large applications where caching demands are extensive.

In practice, Memcached excels when caching frequently accessed data. Typical uses for Memcached include caching key session values and data, which are both accessed often and shared widely. It's also ideal for storing homepage data, which is computationally expensive and frequently accessed, making it crucial for optimal load times.

Caching at a lower level, as with Memcached, effectively reduces load on databases, which often constitute the main performance bottleneck in backend systems. By handling many database requests at the memory level, Memcached accelerates response times and offloads work from the database.

Memcached employs a simple invalidation strategy by setting expiration times on cached items, allowing data to automatically expire rather than requiring manual deletions. This approach can result in slightly outdated data, which is acceptable for summaries, overviews, and other low-criticality pages. For high-sensitivity data, however, it's possible to set up conditional expiration.

Although it reduces database requests, each Memcached call still has a performance cost. To mitigate this, techniques like multi-get can retrieve multiple keys in a single call, reducing response time by returning an array of items. Security is another consideration, as early versions of Memcached had no built-in authentication. With the addition of the SASL Auth Protocol, securing access to Memcached has become easier.

2.4 Columnar database

A columnar database stores data in columns rather than rows. This approach is optimized for Online Analytical Processing and data mining tasks, where efficient read operations over large datasets are essential.

In row-oriented databases, modifying a record is straightforward, but querying might involve reading unnecessary data. Columnar databases, however, allow for reading only the relevant columns. However, writing entire tuples requires multiple column accesses, making columnar databases more suitable for scenarios with high read and lower write demands.

Advantages	**Disadvantages**
Data compression	Increased disk seek time
Improved bandwidth utilization	Increased cost of inserts
Improved code pipelining	Increased tuple reconstruction costs
Improved cache locality	

When tuples need to be analyzed, they are often reconstructed using a large prefetch, which helps minimize the effect of disk seeks across columns.

Compression Columnar databases leverage compression techniques more effectively than row-based databases. These databases take advantage of higher data value locality in columns, enabling advanced techniques like run-length encoding. Additional space can be used to store multiple copies of data in different sort orders, further optimizing query performance.

2.4.1 Cassandra

Originally developed by Facebook and now maintained by the Apache Foundation, Cassandra is a popular column-oriented, NoSQL database widely used for high-throughput applications.

2.4.1.1 Architecture

Cassandra's architecture is optimized for high availability, scalability, and write-intensive workloads, making it ideal for distributed, flexible data storage systems. Its column-oriented design, combined with a masterless structure, eliminates single points of failure while providing tunable consistency for varying application requirements.

Cassandra's masterless architecture ensures no single node holds ultimate control, distributing data across a ring of nodes. This ring topology connects servers in a circular fashion, enabling redundancy and fault tolerance. Each node in the ring is responsible for a portion of the data, and data placement follows a clockwise strategy to the next nodes in the ring, ensuring even distribution.

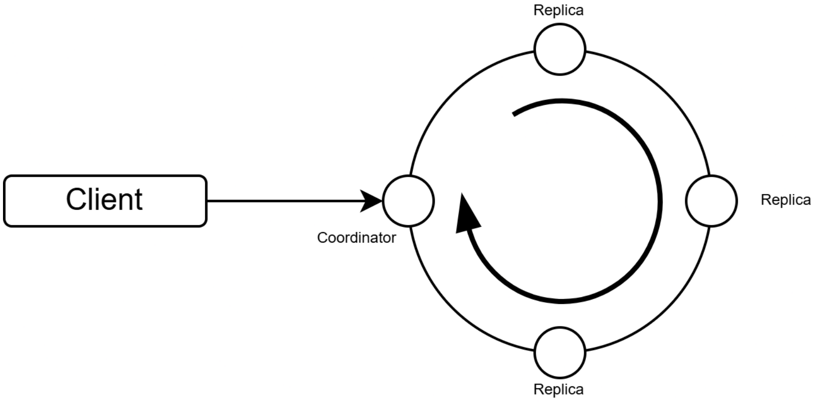


Figure 2.4: Ring architecture

Cassandra minimizes network overhead, except during rare gossip storms. Nodes periodically exchange small membership updates via the gossip protocol, ensuring consistent and accurate cluster state awareness. Each node gossips with up to three peers, fostering rapid convergence and anti-entropy to synchronize data efficiently.

Gossip protocol Cassandra uses a lightweight gossip protocol to maintain cluster state while minimizing bandwidth consumption. Each node communicates with a small subset of peers (up to three) during a gossip round. This exchange helps disseminate cluster information efficiently and introduces a layer of anti-entropy, allowing data to converge more quickly across the cluster. Nodes periodically share their membership list, updating their local view upon receiving updates from peers. This ensures the cluster remains synchronized and resilient, even in the face of failures or network issues.

The operations that we can do are:

- *Write*: Cassandra prioritizes speed and availability for writes, avoiding disk seeks and locks to maintain scalability. When a client sends a write request, it is directed to a coordinator node, which identifies responsible replicas and forwards the data. If a replica is temporarily unavailable, the coordinator buffers the data, ensuring eventual consistency via a mechanism called hinted handoff. Data is initially written to a commit log for durability, then updated in an in-memory structure called the memtable. Over time, the memtable is flushed to disk as SSTables, which are indexed and include bloom filters for efficient reads. Periodic compaction merges updates and applies tombstones to optimize storage.

A bloom filter is a compact and efficient data structure used to represent a set of items. It allows quick membership checks to determine if an item is possibly in the set, with minimal memory overhead. While bloom filters can produce false positives they never produce false negatives, ensuring that if an item is in the set, it is always identified as such. This makes bloom filters highly effective for applications where occasional false positives are acceptable, but false negatives are not.

- *Delete*: deletions are handled lazily by adding a tombstone marker rather than immediately removing the data. These tombstones are processed during compaction to permanently delete the marked records, reducing overhead and improving write performance.
- *Read*: reads often involve querying the closest replica for data, but the coordinator may contact multiple replicas to ensure consistency. If discrepancies arise between replicas, the system initiates a background process called read repair to synchronize the data. While reads may be slower than writes due to consistency checks, they remain efficient through the use of memtables, commit logs, and bloom filters to limit disk access.

Consistency levels Cassandra provides flexible consistency levels, allowing clients to choose based on application needs:

- `ANY`: data can be written to any node (even non-replicas).
- `ONE`: at least one replica must confirm.
- `QUORUM`: a majority of replicas (across all datacenters) must confirm.
- `LOCAL_QUORUM`: majority confirmation in the coordinator's datacenter.
- `EACH_QUORUM`: majority confirmation in each datacenter.
- `ALL`: all replicas in all datacenters must confirm.

2.4.1.2 Data model

In Cassandra, data is organized into column families, which are analogous to tables in SQL but are more flexible and can have unstructured, client-specified schemas. Column families allow the storage of sparse data, where some columns may be missing in specific rows, fitting Cassandra's NoSQL model.

Each Cassandra keyspace functions similarly to a database, typically used per application with certain configurations set per keyspace. The primary elements in Cassandra's data model include:

1. *Keyspace*: equivalent to a database, typically unique per application.
2. *Column family*: groups records of similar types, stored as sparse tables.
3. *Columns*: each column has three parts:

- *Name*: a byte array used for sorting, querying, and indexing.
- *Value*: a byte array; typically not queried directly.
- *Timestamp*: used for conflict resolution, with the most recent write winning.

Additionally, Cassandra supports super columns, which group columns under a common name but lack indexing for sub-columns. These are often used to denormalize data from standard column families.

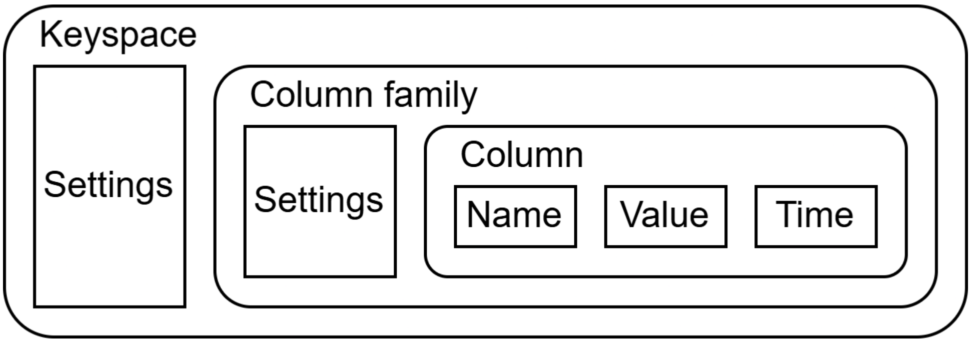


Figure 2.5: Cassandra data model

2.4.1.3 Query language

To interact with Cassandra, developers can use the API for various read and write operations:

```
// retrieve a specific column at the given path
get(): Column
// retrieve a set of columns in one row specified by the slice predicate
get_slice(): List<Column>
// retrieve slices for multiple keys based on a SlicePredicate
multiget_slice(): Map<key, List<Column>>
// retrieve multiple columns according to a specified range
get_range_slices(): List<KeySlice>
```

For writing operations, Cassandra provides commands such as:

```
// insert a new element in a column
client.insert()
// update an existing element in a column
batch_mutate()
// remove an existing element from a column
remove()
```

CQL Cassandra also supports SQL since it is based in tables with a single column. The main difference is that in relational databases we have a domain-based model, while in columnar databases we have a query-based model. Thus, in this case we start from the queries, and then we design the data model based on that since each column has a key that is used to filter all the elements in the column.

Rows are distributed across the cluster based on the partitioning strategy:

- *Random partitioning*: uses the hash of the row key for uniform data distribution.

- *Order-preserving partitioning*: uses the actual row key to maintain a natural order.

To create a keyspace, use the following command:

```
CREATE KEYSPACE identifier
WITH properties
```

A table can have multiple clustering and partition keys. The first set of values is called the partition key, and it determines how the data is distributed across the Cassandra nodes. The second set of values is the clustering key, which defines how the data is stored within each partition, specifically the sorting order.

```
PRIMARY KEY ((partition_key, ...), clustering_key, ...)
```

When creating a table, you can use Clustering Keys to specify the order in which data is stored.

```
CREATE TABLE table_name (column_name column_type, ...)
WITH CLUSTERING ORDER BY (key ASC, ...)
```

To verify whether a keyspace or table has been successfully created, you can use the `describe` command. t can also be applied to check other elements:

```
DESCRIBE keyspaces
```

Before performing operations on tables, you need to specify the keyspace you're working with:

```
USE keyspace_name
```

Indexes are crucial in Cassandra tables as they allow for efficient querying of columns. While this advantage might not be obvious with small datasets, it becomes essential when dealing with larger datasets. To create a secondary index, use the following command:

```
CREATE INDEX identifier
ON table_name(column_name)
```

2.5 Information retrieval database

Information retrieval databases adopt a search-engine-oriented architecture rather than a traditional database approach. A prominent example is the ELK stack, which is structured into three key components:

- *Elasticsearch*: serves as the core of the stack and functions as a robust search and analytics engine. It provides near real-time indexing, meaning documents become searchable shortly after being indexed. Built on Apache Lucene, Elasticsearch enables full-text search capabilities, supports a distributed architecture for scalability and reliability, and uses a RESTful interface for easy integration with external systems.

- *Logstash*: streaming ETL (Extract, Transform, Load) engine of the stack. It facilitates centralized data collection, processing, and real-time enrichment. Logstash is data agnostic, capable of handling various data formats, and supports a wide range of integrations and processors.

- *Kibana*: complements Elasticsearch by offering an open-source data visualization dashboard. It allows users to create visual representations of the data indexed in Elasticsearch through an intuitive and straightforward interface. Despite its simplicity, Kibana is highly customizable, enabling the creation of detailed and complex visualizations to suit specific needs.

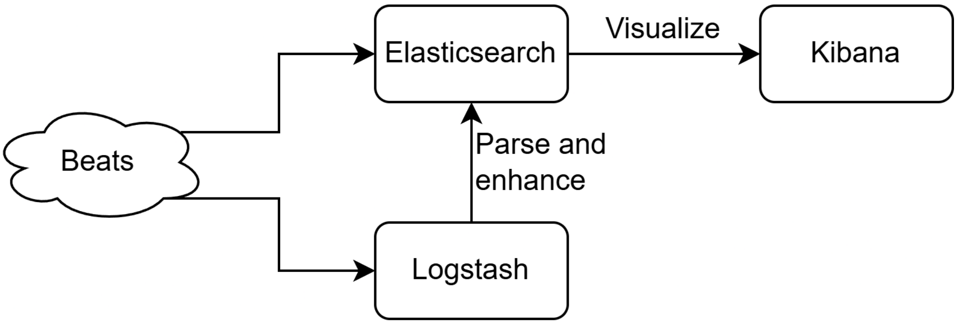


Figure 2.6: ELK stack

2.5.1 Elasticsearch

Elasticsearch introduces two features that are not typically found in traditional databases:

- *Relevance*: relevance refers to how Elasticsearch handles query results. In relational databases, a query retrieves data that exactly matches the specified conditions, returning a definitive answer. In contrast, Elasticsearch prioritizes returning the best-matching elements rather than every possible match. This is made possible through its use of an inverted index, which lists every unique word in all documents and maps each word to the documents in which it appears.

  Elasticsearch also organizes data using indexes, which define the structure of documents through mappings. Each index is divided into shards, which help distribute operations across nodes to improve performance and resilience to faults. Shards are further replicated into replicas, ensuring data redundancy by storing copies on different nodes for fault tolerance.

- *Ranking*: unlike traditional databases, which often focus on returning all matches without prioritization, Elasticsearch determines the best and worst results for a query by calculating a relevance score. This ranking system ensures that the most relevant elements are highlighted, making Elasticsearch particularly effective for search applications.

  Elasticsearch calculates the relevance of search results using Lucene's Practical Scoring Function, which incorporates TF-IDF (Term Frequency-Inverse Document Frequency). TF-IDF is a statistical measure used to evaluate how important a term is within a document relative to a collection of documents. This scoring method ensures that results are ranked according to their relevance to the query. We have the follwing elements:

  – *Term frequency* (TF): the frequency of a term i in a document j, normalized by the total number of terms in the document:

  $$\mathrm{tf}_{i,j} = \frac{n_{i,j}}{|d_j|}$$

  Here, $n_{i,j}$ is the count of term i in document j, and $|d_j|$ is the total number of terms in document j.

  - *Inverse document frequency* (IDF): this measures how unique or common a term is across the entire collection of documents. Terms that appear in many documents are considered less significant. The formula is:

    $$\mathrm{idf}_i = \log \frac{|D|}{|\{d \mid i \in d\}|}$$

    Here, $|D|$ is the total number of documents, and $|\{d \mid i \in d\}|$ is the number of documents containing the term i.

The final TF-IDF score for a term i in document j is computed as the product of term frequency and inverse document frequency:

$$(\text{tf-idf})_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i$$

Mapping A mapping in Elasticsearch defines the structure and types of fields within an index. It determines: which fields are searchable, and support for full-text search, as well as time-based and geo-based queries. However, mapping on an existing index cannot be changed once documents are stored. By default, Elasticsearch attempts to infer the structure of documents through dynamic mapping, which can be risky.

2.5.1.1 Data model

Elasticsearch stores data as JSON documents, making it easy to integrate with web applications. These documents are:

- *Distributed*: data can be accessed from any node within a cluster.
- *Indexed*: newly stored documents are immediately indexed and made fully searchable.

2.5.1.2 Query language

Interaction with Elasticsearch is achieved by sending requests to REST endpoints, with actions determined by the following HTTP verbs:

- `GET`: retrieve documents or index metadata.
- `POST/PUT`: create new documents or indices.
  - `POST`: does not require an ID; Elasticsearch auto-generates one.
  - `PUT`: requires an ID and is used to create or update a specific document.
- `DELETE`: remove documents or indices.

Requests can be sent using: command-line tools, software like Postman, or developer tools in Kibana.

Indexing and mapping Indices and mappings are defined as follows:

```
PUT /index_name
```

We can define a mapping in the following way:

```
PUT /index_name/_mapping
```

Common field types include:

- *Date*: for timestamps; formats can be specified.
- *Keyword*: for structured data like emails, tags, and postcodes.
- *Long*: for 64-bit integers.
- *Text*: for full-text search, with customizable analyzers to preprocess data.

Language analyzer Language analyzers preprocess text for search by: removing stopwords, and performing stemming to reduce words to their root forms.

```
POST /_analyze
{
    text
}
```

This returns a JSON structure optimized for the given input.

Search To retrieve a document:

```
GET /index_name/type_name/id
```

To perform a query:

```
GET /index_name/_search
{
    "query": {
        "match": {
            conditions
        }
    }
}
```

To count matching documents:

```
GET /index_name/_count
{
    "query": {
        "match": {
            conditions
        }
    }
}
```

Elasticsearch supports filters for exact matches, which are used for unanalyzed fields, do not calculate relevance (binary match), and are cacheable for performance. Example of a query with filters:

```
GET /index_name/_search
{
    "query": {
        "bool": {
            "filter": {
                "term": { "field": "value" }
            }
        }
    }
}
```

Filters can be combined with queries to enhance efficiency and relevance scoring at query time. Elasticsearch also supports searching across multiple indices simultaneously. The logical operators are must (AND), must not (NOT), and should (OR)

Elasticsearch provides a powerful aggregation framework to analyze data. Aggregations can be defined as follows:

```
GET /index_name/_search
{
    "query": {
        "size": 0,
        "aggs": {
            "aggregation_name": {
                "type": { "field": "field_name" }
            }
        }
    }
}
```

Supported types of aggregations include:

- *Metrics*: summarize numeric data.
- *Buckets*: group data into categories.
- *Pipeline*: process results of other aggregations.
- *Matrix*: perform advanced mathematical operations.

By combining queries, filters, and aggregations, Elasticsearch enables powerful data retrieval and analysis.

2.5.2 Logstash

Beats is a lightweight platform designed to serve as a data shipper, collecting and sending logs and metrics from hosts or containers to systems like Logstash or Elasticsearch. It includes several specialized modules, such as Filebeat for log file collection, Metricbeat for system and service metrics, Packetbeat for network data capture, and Heartbeat for monitoring service availability. These modules focus on data collection and shipping, while Logstash handles the more complex tasks of processing, structuring, normalizing, and enriching the data. Logstash can also receive data from sources where Beats are not deployed, supporting protocols like TCP, UDP, HTTP, and pool-based inputs like JDBC.

Processing Logstash uses filter plugins (or processors) for data wrangling. These filters help structure, normalize, and enrich incoming data, enabling users to build sophisticated data pipelines. These pipelines can be tailored to meet specific data processing needs, allowing for flexible and efficient management of diverse data types. After processing, Logstash can emit data to Elasticsearch or other data stores, using output plugins that support protocols such as TCP, UDP, and HTTP.

Modules Logstash and Beats provide modules that facilitate automated processing for specific data types. These modules handle tasks like automated parsing and enrichment within Logstash, the creation of custom schemas in Elasticsearch, and the generation of default Kibana dashboards for data visualization. These modules help streamline the process of transforming raw data into insightful visual representations, making it easier to integrate with Kibana for analysis.

Data flow The flow of data in Logstash begins with the input, where data is ingested from various sources. It then passes through one or more filters, where it is structured, normalized, and enriched according to predefined rules. Finally, the processed data is emitted to the specified output, which could be Elasticsearch or another data store, or it could be sent to external systems via various protocols. This pipeline model ensures that data is efficiently managed, processed, and routed to the appropriate destination for further use or analysis.

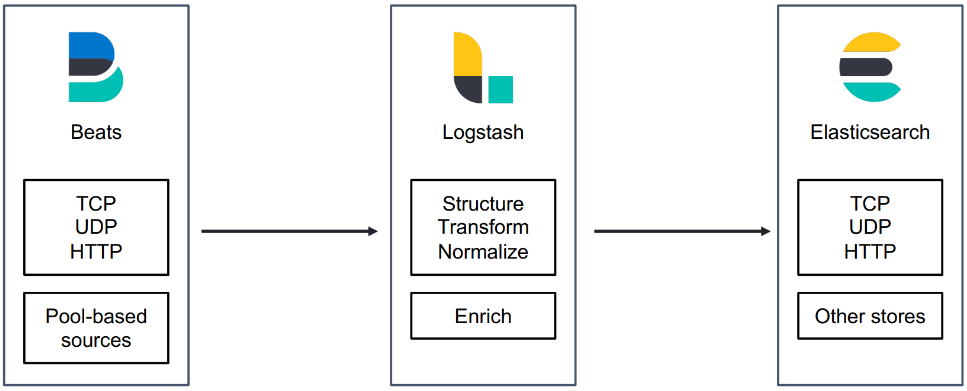


Figure 2.7: Logstash data flow

2.5.2.1 Architecture

In Logstash, the event is the primary unit of data, similar to a JSON document. These events flow through pipelines, which define the logical flow of data from ingestion to processing and output. A pipeline can handle multiple inputs simultaneously, managing the flow of data through a single queue, which can be configured to be either in-memory or persistent. This buffer ensures that data is processed in an organized manner, even under heavy load. Logstash employs workers to process the data, ensuring scalability by allowing multiple parallel processes to handle the incoming data efficiently.

Logstash instances can support multiple pipelines, each handling separate data flows independently. This makes it possible to configure different pipelines for distinct tasks within the same Logstash process.

To modify how data is represented, Logstash uses codecs to handle serialization and deserialization. Codecs are applied during data input or output, enabling flexibility in how data is processed as it moves through the system.

Logstash also employs an at least once message delivery strategy, which ensures that messages are generally delivered exactly once. However, in the case of an unclean shutdown, duplicates may occur. To prevent data loss, Logstash utilizes a Dead Letter Queue for events that fail to be processed. This mechanism allows undeliverable events to be stored temporarily, freeing resources in the pipeline and ensuring that subsequent events are processed without delay.

2.5.3 Kibana

Kibana is an open-source data visualization tool designed specifically for Elasticsearch. It allows users to create interactive and dynamic visualizations based on the content indexed in an Elasticsearch cluster. While Kibana is intuitive and easy to use at first, it also offers a high degree of customization, enabling users to build complex and detailed representations of their data. Regardless of the subscription level, Kibana provides robust capabilities to aggregate, organize, filter, and visualize data in various formats. Kibana supports a wide array of visualizations, including charts, graphs, maps, and more. One of Kibana's most powerful features is the ability to create custom dashboards that integrate multiple data types and representations.

2.6 Vector database

Vector databases are gaining significant traction, driven by the rise of artificial intelligence and Machine Learning applications, particularly those leveraging large language models in combination with retrieval-augmented generation. These technologies enable more sophisticated data retrieval methods, providing powerful enhancements to search, recommendation, and discovery systems.

By storing and processing data as high-dimensional vector representations, vector databases facilitate more efficient and context-aware data retrieval. Unlike traditional databases that rely on keyword-based search, vector databases allow AI systems to interpret the semantic similarity and context between data points. This capability is especially valuable for applications that require understanding the meaning and relationships between large datasets, such as personalized search, content recommendations, and natural language understanding.

Vectors A vector database organizes data as high-dimensional vector representations, which allows for contextual search and discovery that goes beyond simple keyword matching. Vectors are grouped according to their semantic similarity, meaning the database can retrieve data based on the inherent meaning or context, rather than exact matches. This makes data retrieval more efficient and meaningful, particularly for tasks that involve understanding the relationship between data points.

In the fields of mathematics and physics, a vector is a quantity characterized by both magnitude and direction. Similarly, in vector databases, vectors represent various attributes or features of an object, capturing key characteristics in a structured, multidimensional space. These vector representations form the foundation of information processing in many AI and Machine Learning applications, enabling machines to understand and interpret complex data patterns.

One common technique used to create these vectors is embedding, which transforms data into vector format. For textual data, pre-trained models map words and phrases into a multidimensional space, effectively capturing their meanings and relationships in context. This process allows vector databases to find and retrieve relevant information based on vector similarity, rather than relying on exact text matches, offering more nuanced and context-aware search results.

Similarity o measure the similarity between vectors, metrics like Cosine similarity and Euclidean distance are commonly employed. Cosine similarity is particularly popular due to its computational efficiency and ability to capture the orientation of vectors in space, making it ideal for comparing the semantic similarity between high-dimensional vectors. These similarity measures enable the database to determine how closely related two data points are, helping AI systems retrieve the most contextually relevant information based on the vector representations.

Vectors matching Vector databases excel not only in efficiently storing high-dimensional vectors but also in enabling fast and accurate matching of vectors. Searching for nearest neighbors in an unindexed database involves calculating the distance from the query vector to each point in the dataset, which can be highly computationally expensive for large datasets. Therefore, using indexing techniques to speed up this process is essential for efficient vector matching.

Tree-based indexes One common approach to creating efficient vector indexes is through tree-based techniques, which organize data into structures that optimize search operations. Below are two popular tree-based indexing methods used for vector matching:

- *K-dimensional trees*: a type of binary space partitioning tree. In this structure, each node represents a k-dimensional point. The non-leaf nodes define splitting hyperplanes that partition the space into two regions. Each split is based on the median value of a specific dimension. The primary advantage of K-d trees is that they reduce the search space by narrowing down potential matches at each level of the tree. The time complexity for search operations in K-d trees is $\mathcal{O}(\log n)$, where n is the number of objects in the dataset. However, this method becomes less efficient as the dimensionality of the data increases. With high-dimensional vectors, the tree becomes less effective, and the performance may degrade to a level similar to exhaustive search, as most of the points in the tree will need to be evaluated.

- *R-trees*: this indexing structure groups nearby objects into bounding rectangles, which are then represented at higher levels of the tree. The key idea is that if a query does not intersect a bounding rectangle at a certain level, it will not intersect any of the objects contained within it. This enables the search process to eliminate large portions of the dataset without checking each individual vector. At the leaf level, each rectangle corresponds to a single object, while higher levels aggregate multiple objects into larger rectangles, forming a coarse approximation of the dataset. R-trees are balanced binary trees, meaning that the leaf nodes are always at the same depth. The tree is structured in pages, and each page has a minimum and maximum fill limit, which optimizes storage and retrieval. A key challenge in R-tree construction is ensuring that the tree is well-balanced, that the bounding rectangles do not cover too much empty space, and that there is minimal overlap between them. This reduces the number of subtrees that need to be examined during a search, improving search efficiency.

Approximation For high-dimensional data, exact search techniques like K-d trees and R-trees often perform poorly due to the curse of dimensionality. To address this challenge, vector databases use Approximate Nearest Neighbor (ANN) algorithms to speed up searches in large datasets.

- *Locality Sensitive hashing* (LSH): LSH is a technique that hashes similar items into the same buckets, increasing the likelihood of matching similar data points. Unlike traditional hashing, which minimizes collisions, LSH maximizes them to reduce the dimensionality of data, making it suitable for clustering and nearest-neighbor searches.

- *Hierarchical Navigable Small World* (HNSW): HNSW is an efficient algorithm for ANN searches in high-dimensional spaces. It organizes data points into a hierarchical graph where nodes represent vectors. The search begins at the top layer, which contains a few representative nodes, and moves downward through more detailed layers, progressively refining the result. HNSW sacrifices some accuracy for faster search times, making it highly scalable, fast, and adaptable to large datasets. It is also memory efficient, suitable for real-time applications with dynamic data.

- *Inverted File with Product Quantization* (IVFPQ): algorithm designed for efficient Approximate Nearest Neighbor (ANN) search in high-dimensional vector spaces, particularly for large datasets. It balances speed and memory efficiency by combining two key techniques: coarse quantization and product quantization.

  At its core, IVFPQ organizes the dataset using an inverted index, a structure that maps data points to representative clusters. This organization enables the algorithm to limit its search to specific regions of the vector space, significantly reducing the computational workload. The process begins with coarse quantization, where a clustering algorithm such as k-means divides the dataset into clusters. Each vector is assigned to the nearest cluster center, known as a coarse centroid. These centroids serve as bins, grouping similar vectors together. When a query is presented, the algorithm identifies the relevant clusters instead of searching the entire dataset.

  Once the clusters are narrowed down, product quantization comes into play. This technique compresses the vectors within each cluster by dividing them into smaller sub-vectors, each of which is quantized separately. This compression dramatically reduces storage requirements while maintaining the ability to approximate vector similarities. The query vector is then compared to these compressed representations, allowing the algorithm to estimate distances efficiently. However, as with any compression method, the reconstructed vectors differ slightly from the originals, introducing a small loss in accuracy.

  The IVFPQ search process operates in two stages: a coarse search to locate relevant clusters and a refined search within those clusters to approximate the nearest neighbors using product quantization. While this method doesn't guarantee finding the exact nearest neighbors, it offers a highly efficient and scalable solution for ANN searches, especially in scenarios involving massive datasets.

2.6.1 FAISS

Faiss, developed by Facebook AI, is a powerful library designed for efficient similarity search and clustering in high-dimensional vector spaces. It excels at Approximate Nearest Neighbor search, enabling users to find vectors similar to a query vector within an indexed dataset. Highly optimized for performance and accuracy, Faiss is a popular choice for applications involving large-scale data.

Faiss provides various indexing structures to optimize searches for datasets of different sizes and complexities. These include:

- *Flat index*: stores all vectors directly for exact nearest neighbor searches.
- *Inverted File Index*: clusters vectors to narrow down search space, often used with product quantization to handle larger datasets.
- *FHierarchical Navigable Small World*: a graph-based structure that enables fast ANN searches through hierarchical relationships between vectors.

In addition to indexing, Faiss supports clustering, allowing users to group high-dimensional data efficiently. This is useful for tasks like vector quantization, document clustering, and unsupervised learning.

Search The workflow in Faiss typically follows three steps:

1. *Indexing*: users create an index tailored to their dataset and performance requirements, choosing from various structures like Flat, IVF, or HNSW.
2. *Quantization*: for large datasets, Faiss can apply compression techniques like product quantization, reducing memory usage and speeding up searches.
3. *Search*: once indexed, query vectors are compared against the stored vectors to find the nearest neighbors. Faiss supports approximate searches that prioritize speed while maintaining reasonable accuracy.

Faiss is designed for scalability, capable of handling millions or even billions of vectors. It leverages GPU acceleration and advanced indexing to perform searches in milliseconds, even on massive datasets. The library is highly flexible, offering numerous customization options for indexing and search algorithms. As an open-source and cross-platform tool, it integrates easily with popular programming languages like Python and C++.

Implementation Faiss works with collections of fixed-dimensional vectors, typically in the range of tens to hundreds of dimensions. Data is stored in 32-bit floating-point matrices with row-major order, where each row corresponds to a vector. This format ensures compatibility and consistency during indexing and search operations.

2.6.2 Pinecone

Pinecone is a vector database designed for efficient similarity search and large-scale production environments. Unlike standalone vector indexes, Pinecone combines indexing, querying, and post-processing with advanced features like distribution, scalability, and fault tolerance to handle complex workflows in real-world applications.

Pinecone organizes vectors using algorithms such as Product Quantization, Locality-Sensitive Hashing, or Hierarchical Navigable Small World. These algorithms structure the data for fast retrieval, enabling efficient nearest neighbor searches. When a query vector is submitted, Pinecone compares it against the indexed vectors in the database using a similarity metric associated with the chosen indexing algorithm. This step quickly identifies the most relevant vectors. After the initial search, Pinecone applies additional processing to refine the results. This could include re-ranking the nearest neighbors using a different similarity measure or applying custom filters to meet specific requirements.

Pinecone excels in handling large-scale vector search tasks by offering a distributed, fault-tolerant architecture that ensures reliability and performance. Its sharding and replication strategies enable efficient query handling and high availability. Additionally, its integration of advanced indexing techniques with post-processing makes it ideal for applications that demand precision and scalability.

2.6.3 Zilliz

Zilliz is a technology company specializing in unstructured data management and AI-powered analytics. Its mission is to revolutionize how enterprises handle large, high-dimensional datasets such as images, audio, video, and text. By providing scalable and efficient data processing solutions, Zilliz empowers organizations to leverage AI and Machine Learning for smarter, data-driven decision-making.

Through innovative technologies and tools, Zilliz addresses the challenges of managing and analyzing unstructured data at scale, enabling businesses to unlock the full potential of their data assets in AI and advanced analytics workflows.

2.6.3.1 Milvus

Milvus, developed by Zilliz, is an open-source vector database designed to handle large-scale, high-dimensional data for similarity search and unstructured data retrieval. It integrates seamlessly with AI models and applications, making it ideal for industries such as e-commerce, healthcare, and finance, where large datasets and rapid processing are critical.

Milvus is optimized for scalability, offering performance that can range from small projects to large, enterprise-level deployments. The architecture of Milvus is built around several key principles, including disaggregated storage and computation, microservices, and separation of streaming and historical data. This flexible design allows Milvus to scale efficiently based on specific use cases and workloads.

One of Milvus' standout features is its disaggregated architecture, which uses separate components for storage and computation, making it adaptable to various deployment environments. It supports pluggable engines, storage solutions, and indexing techniques, allowing users to select configurations that best suit their needs. Milvus also relies on a logging mechanism called Log As Data, ensuring that all data changes are tracked in real-time, facilitating efficient updates and retrieval.

In terms of data management, Milvus uses a shard-based system. Each shard is overseen by a supervisor, known as the shard leader, who ensures that new data is added, stored safely in object storage, and kept up to date for search requests. When historical data is needed, the supervisor forwards the request to other query nodes, ensuring seamless access to both current and past information.

Milvus uses two types of data segments to manage the flow of information. The Growing Segment handles in-memory data, replaying information from the Log Broker, and ensures that data is fresh and appendable through a flat indexing method. Meanwhile, the Sealed Segment is immutable and uses alternative indexing methods for optimized search and retrieval, making it more efficient for large-scale data storage.

To maintain data integrity, Milvus utilizes a Write-Ahead Log, which temporarily holds new data before it is fully integrated into the system. This log functions like a to-do list for the database, ensuring that all new data is processed and recorded efficiently before being committed to the permanent storage.

With its combination of flexible architecture, scalable components, and advanced data management techniques, Milvus is an effective solution for organizations that need to store, search, and retrieve massive amounts of unstructured data with high efficiency.

2.6.4 Neo4j

Neo4j has integrated native vector search into its core capabilities, enabling powerful semantic graph search. This integration paves the way for neurosymbolic AI, where the strengths of symbolic reasoning and Machine Learning are combined to produce more intelligent systems. By combining vector search with traditional graph-based querying, Neo4j enhances the ability to extract richer insights from data, which is especially useful in applications involving generative AI and semantic search.

The inclusion of vector search allows Neo4j to serve as a robust foundation for long-term memory in large language models, helping to reduce errors like hallucinations that can occur in AI-generated outputs. This integration not only improves the accuracy of results but also supports more advanced AI applications by enabling a deeper, context-aware understanding of data relationships.

CHAPTER 3

Data storage and processing

3.1 Machine Learning operations

A feature store is a centralized repository designed to store, manage, and serve features for Machine Learning models. Features are the input variables that describe the data objects used by models to make predictions. The main goal of a feature store is to simplify the process of creating, sharing, and using these features across various Machine Learning models. This ensures that features are easily accessible, consistent, and reusable, making it easier to develop, maintain, and deploy Machine Learning systems. Feature stores are a key component of Machine Learning operations.

Machine Learning operations refers to a set of practices and tools that streamline the development, deployment, and maintenance of Machine Learning models in production. It extends DevOps principles to the Machine Learning world, focusing on the unique challenges of building, deploying, and monitoring Machine Learning systems at scale. Machine Learning operations brings together data scientists, Machine Learning engineers, and operations teams, enabling them to collaborate effectively to ensure reliable and efficient deployment of Machine Learning models. The Machine Learning operations lifecycle includes the following:

1. *Development*: data scientists experiment with models, algorithms, and feature sets. Automation ensures proper versioning of code, models, and datasets.
2. *Training and validation*: the model is trained and validated using test datasets. Machine Learning operations automates this process, ensuring reproducibility.
3. *Deployment*: once validated, the model is deployed into production using automated practices.
4. *Monitoring*: post-deployment, the model's performance is continuously monitored for accuracy, data drift, and concept drift.
5. *Retraining*: when performance drops or new data is available, models are retrained automatically to stay current.

Machine Learning operations is essential for transitioning Machine Learning from research to scalable production systems. By automating workflows, ensuring consistency, and enhancing collaboration, Machine Learning operations helps organizations efficiently manage and operationalize Machine Learning models at scale.

3.1.1 Feature stores

A feature store is a centralized repository used to manage, store, and serve features for Machine Learning models. Features are the input variables that help Machine Learning models make predictions. The feature store's main goal is to streamline the process of creating, sharing, and using these features across various models, ensuring they are consistent, reusable, and easily accessible. This is particularly important for maintaining high-quality data and enabling collaboration between data scientists, Machine Learning engineers, and data engineers.

A feature store offers consistency, ensuring that features used for training and inference are aligned, which helps improve model accuracy. It also boosts efficiency by enabling feature reuse across models and teams, reducing redundant work and accelerating model development.

Definition (*Feature engineering*)**.** Feature engineering is the process of transforming raw data into meaningful features that improve the performance of Machine Learning models.

This can range from simple transformations, like aggregations, to more complex methods, such as Machine Learning-generated features like word embeddings. Feature engineering's ultimate goal is to create a better dataset that enhances the performance of Machine Learning algorithms.

The role of a feature store is to act as a centralized repository where curated features are stored. It serves as a data management layer, enabling collaboration between teams, and provides an interface that converts raw data into features used in model training and inference. By ensuring consistency across features used in both training and deployment phases, a feature store eliminates discrepancies and reduces the risk of errors. It also prevents duplication of code, making the process more efficient and maintaining alignment between model development and deployment.

Feature stores are essential when deploying models at scale and in production, as they provide consistency and enable teams to collaborate and reuse features. However, for small teams or proof-of-concept projects, the overhead of using a feature store might not be necessary, as simpler workflows can often suffice.

3.1.2 Feast

Feast is an open-source feature store that provides isolation for feature stores at the infrastructure level using resource namespacing. This means that different projects or teams can work with their own feature stores without interfering with each other, allowing for more organized and secure management of features.

For offline use cases that only rely on batch data, Feast does not need to ingest data itself. Instead, it can query existing data directly, making it easy to integrate with other data sources. For online use cases, Feast supports a process called materialization, where features from batch sources are ingested and made available for real-time use. It also supports pushing streaming features to make them available both offline and online.

In Feast, an entity is a group of semantically related features, typically defined to map to the specific domain of a use case. Each feature view is a collection of features, and the entity key is the collection of entities for that view. It is important to reuse entities across feature views to ensure consistency and avoid duplication.

A data source in Feast refers to the raw data that users own, such as a table in BigQuery. Feast does not manage the raw data itself but rather loads this data and performs various operations to retrieve and serve features. Feast uses a time-series data model to interpret data in the underlying data sources. This model is used to build training datasets or materialize features into an online store for real-time use.

Feast utilizes a registry to store all the applied Feast objects, such as feature views and entities. The registry offers methods to manage these objects, including applying, listing, retrieving, and deleting them. This centralized registry allows for better organization and management of features across different projects.

3.2 Data warehouse

Definition (*Online Transaction Processing*)**.** Online Transaction Processing refers to the systems that handle day-to-day transactions at operational sites.

Definition (*Online Analytical Processing*)**.** Online Transaction Processing refers to systems designed to process and analyze data stored in large, integrated data warehouses.

In general, IT systems can be categorized into two types: transactional and analytical. Online Transaction Processing systems typically serve as the source of data for data warehouses, while Online Transaction Processing systems are used to analyze that data.

The challenge in modern data analytics lies in integrating diverse data sources, ensuring the extraction of coherent insights, and handling the heterogeneity and distribution of data. Traditionally, analytics approaches have been divided into two main categories: transactional and analytical. The traditional query-driven approach to analytics tends to be reactive, often lazy, and on-demand, meaning data is accessed only when needed and queries are executed as they arise.

In contrast, the warehousing approach integrates data in advance, storing it in a data warehouse where it can be directly queried and analyzed. It offers high query performance, though it may not always provide the most up-to-date information. Since the data warehouse operates separately from operational systems, it doesn't interfere with local processing. Complex queries are executed within the warehouse, while operational systems continue their tasks without disruption. Additionally, the data stored in the warehouse can be modified, annotated, summarized, and restructured as needed, and historical information can be preserved for analysis. However, the query-driven approach still proves more suitable in certain scenarios.

Definition (*Data warehouse*)**.** A data warehouse is a collection of data that is subject-oriented, integrated, time-varying, and non-volatile, primarily used to aid decision-making within an organization.

The data warehouse is not just a simple repository; it integrates diverse datasets from various sources into a single storage location, allowing for advanced analysis and decision support. Unlike transactional databases, data warehouses are optimized for analytical queries, not day-to-day transactions. They often include a user interface tailored to executives, managers, and analysts, enabling efficient decision-making.

One key characteristic of data warehouses is their ability to handle large volumes of data, often measured in gigabytes or terabytes. These systems are non-volatile, meaning they store historical data that doesn't change frequently. Updates are typically infrequent, with some warehouses being append-only, meaning that new data is added rather than modifying existing data.

Architecture In a single-layer architecture, each data element is stored only once, with no redundancy. A virtual warehouse, on the other hand, presents views over operational databases without physically storing the data. In more common two-layer architectures, real-time data is combined with derived data for analysis. A three-layer architecture includes conceptual views, separating them from the transformation processes that prepare the data for querying.

Decision Support Systems Data warehousing plays a crucial role in Decision Support Systems, which help knowledge workers—such as executives, managers, and analysts. Online Analytical Processing is a vital component of Decision Support Systems, providing the tools to interact with and analyze data stored in the warehouse.

Relational database management systems are often used as servers for data warehouses, though flat files are less common. Online Transaction Processing systems come in two forms: Relational Online Transaction Processing and Multidimensional Online Transaction Processing. ROnline Transaction Processing uses relational databases to store and manage data warehouse content, with specialized middleware to support Online Transaction Processing functionality. Multidimensional Online Transaction Processing, on the other hand, uses array-based storage and offers direct access to multidimensional data, making it highly efficient for complex analytical queries.

Data mart When comparing a data warehouse with data marts, the key difference lies in scope. An enterprise data warehouse is an organization-wide solution that consolidates information across all departments. In contrast, data marts focus on specific departmental needs, such as marketing or sales. While data marts can be rolled out quickly and efficiently, they can create challenges in integration over time, particularly as multiple marts need to be synchronized.

Virtual warehouse A virtual warehouse allows views to be created over operational databases, making it easy to query and analyze data without physically storing it in a central repository. However, materializing selected summary views to optimize query processing can place considerable demands on the operational databases.

3.2.1 Model

In a data warehouse, the typical structure used to organize the data is the star schema, which differs significantly from the class entity relationship diagram often used in transactional systems. The star schema is designed to optimize querying and analysis, providing a straightforward and efficient way to access large volumes of data.

In addition to the star schema, the snowflake schema is another common structure, which involves normalizing the dimensions of the data to reduce redundancy. This schema is often used when there are hierarchical relationships in the data, such as categories or time periods, that benefit from a more organized, normalized approach.

When dealing with hierarchical data, certain operations become essential for effective analysis. One such operation is drill-down, which involves adding one more analysis dimension, typically to disaggregate the data and view it in finer detail.

On the other hand, roll-up is the inverse operation, where one analysis dimension is removed to aggregate the data at a higher level. This might involve summarizing daily sales data into weekly or monthly totals, providing a broader perspective on trends.

Another important concept in data warehousing is the datacube, which enables users to slice and dice the data, examining it from different angles by selecting specific dimensions. The datacube allows for complex multidimensional analysis, making it easier to explore the data and uncover insights by focusing on different combinations of dimensions.

3.2.2 Snowflake

Snowflake is an enterprise-ready cloud data warehouse solution that is designed to automatically scale in order to balance performance and costs efficiently. One of its major advantages is the separation between compute and storage. Unlike other databases that combine the two, Snowflake allows for independent scaling of compute and storage resources. This means you don't need to size your entire system for your largest workload, which helps avoid unnecessary costs. Additionally, Snowflake provides a single place for storing all types of data, making it easier to manage and analyze large volumes of information.

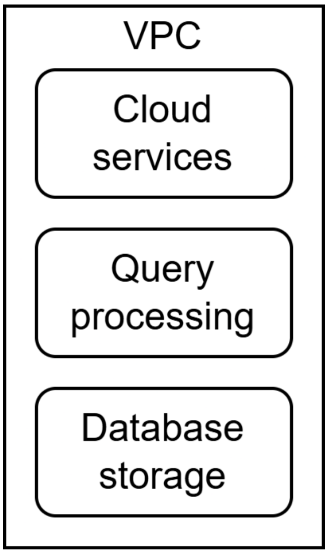


Figure 3.1: Snowflake architecture

The architecture of Snowflake is designed to optimize performance, scalability, and flexibility. It consists of three main layers:

1. *Database storage*: Snowflake organizes data in an internal, optimized, compressed, columnar format. The system handles all aspects of how this data is stored, including its organization, file size, structure, compression, and metadata management. Data is visible or accessible by users only through SQL queries executed within Snowflake.
2. *Query processing*: Snowflake uses virtual warehouses to process queries. These warehouses are based on a Massively Parallel Processing compute cluster, which is made up of multiple compute nodes allocated by Snowflake from a cloud provider. Importantly, virtual warehouses are independent compute clusters that do not share resources with each other. This isolation ensures that the performance of one virtual warehouse does not impact others.
3. *Cloud services*: this layer manages activities across Snowflake, coordinating user requests from login to query dispatch.

Data ingestion Snowflake supports two primary methods for ingesting data:

- *Bulk load*: Snowflake provides the `copy` command for batch loading of data. This command uses the computing resources of the virtual warehouse. The process is managed manually and supports basic transformations such as reordering and excluding columns, changing data types, and truncating strings.

- *Continuous load*: for streaming data, Snowflake offers Snowpipe, which automatically scales up or down depending on the workload. Snowpipe does not use virtual warehouse computing resources, making it ideal for continuous data loading.

Data staging Before data can be loaded into Snowflake for processing, it often needs to be staged. The staging area is an intermediate, transient location used to store and process data before extraction, transformation, and loading. Traditionally, data is staged in a bucket, which provides long-term, cost-effective storage for raw data. Data can also be staged on a local file system for convenience.

3.3 Spark

The main objectives of data processing with Spark are to enable faster and better decisions by supporting a range of use cases:

- *Low latency queries on historical data*: Spark aims to allow for interactive queries on historical data, enabling users to make decisions quickly based on past information.
- *Low Latency queries on live data*: Spark also supports real-time data processing, allowing decisions to be made on live, streaming data as it flows into the system.
- *Sophisticated data processing*: Spark enables the execution of more complex, advanced algorithms, which can result in deeper insights and better decision-making.

The goal is to provide comprehensive support for batch, streaming, and interactive computations, and to make it easy to combine these types of processing in a single workflow. This approach simplifies the development of sophisticated algorithms that can handle various forms of data in a flexible manner.

In modern analytics, many organizations face significant challenges when trying to manage multiple processing stacks.

3.3.1 Hadoop

Hadoop is responsible for storing data across a cluster, with data files split into blocks and distributed across the nodes. These blocks are replicated multiple times for redundancy, providing reliable storage for large amounts of data. HDFS (Hadoop Distributed File System) is optimized for streaming reads of large files, typically 100MB or more, and is designed for a write-once-read-many access model.

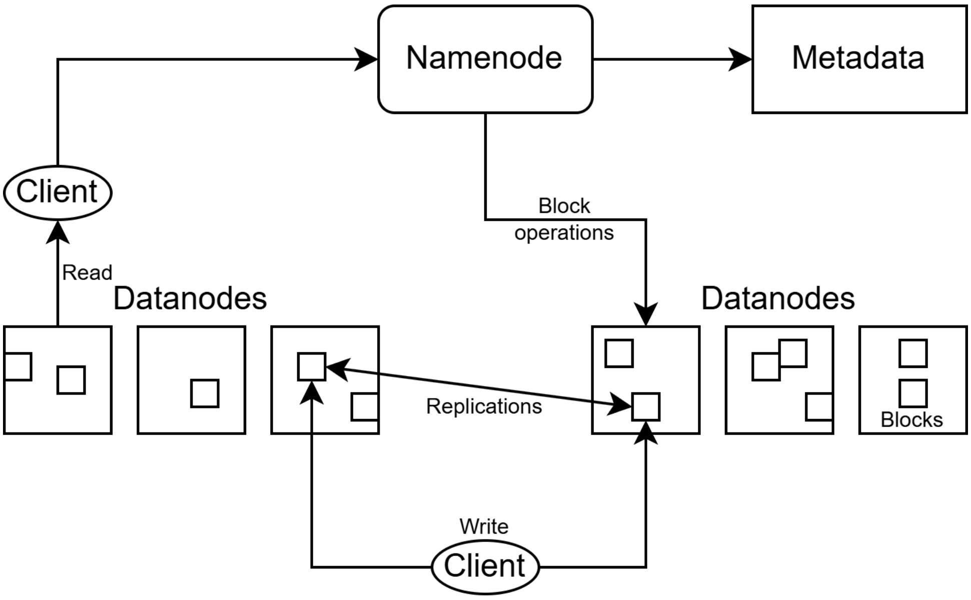


Figure 3.2: Hadoop architecture

The key components and features are:

- *NameNode*: manages the file system namespace, keeps track of where each file's blocks are stored, and handles cluster configuration and replication. It maintains metadata such as the list of files, blocks, and DataNodes.

- *DataNode*: stores the actual data and metadata of blocks, serves data to clients, and periodically reports block information to the NameNode. It also facilitates data pipelining, where blocks are forwarded to other DataNodes.

- *Block replication*: blocks are replicated across multiple DataNodes to ensure data availability. A typical strategy places one replica on the local node, another on a remote rack, and a third on the same remote rack.

- *Fault tolerance*: the system uses heartbeats to detect DataNode failures, and the NameNode will replicate blocks to other DataNodes if a failure is detected. Clients can access data directly from the nearest replica, ensuring fast data retrieval.

- *Metadata and data consistency*: the NameNode stores metadata in memory, and DataNodes use checksums to validate data integrity. In case of NameNode failure, transaction logs are stored in multiple locations to prevent data loss.

- *Cluster maintenance*: Hadoop includes a Rebalancer to balance disk usage across DataNodes and a Secondary NameNode to merge transaction logs and maintain the file system's integrity.

- *Authentication and security*: Hadoop supports two authentication mechanisms: simple OS-based authentication and more secure Kerberos authentication. File and directory permissions are similar to POSIX standards, with access control lists allowing for fine-grained permissions.

3.3.2 Spark

Apache Spark is an open-source project that has gained tremendous popularity since its first release in February 2013. It quickly became a go-to solution for big data processing due to its ease of use and speed. Spark was originally created at the AMPLab at the University of California, Berkeley, and has since become a core part of the modern data analytics ecosystem.

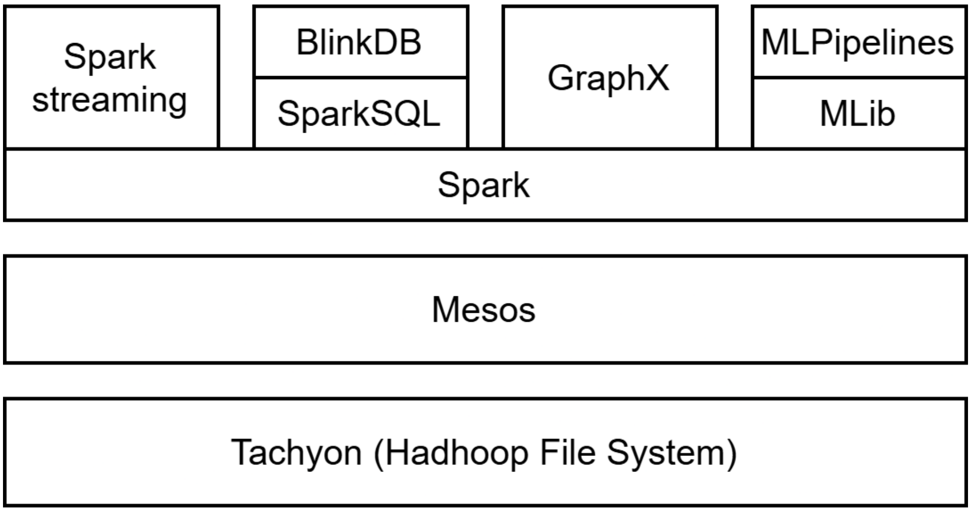


Figure 3.3: Spark architecture

Spark is a powerful, fault-tolerant, in-memory storage system using Resilient Distributed Datasets. It offers fast performance and is easy to use, with 5-10x less code than Hadoop. It supports a variety of programming models, including interactive and iterative applications, and is scalable across clusters. The key features are:

- *Fault-tolerant*: Spark handles failures efficiently by tracking the lineage of Resilient Distributed Datasets.
- *Unified programming models*: Spark supports SQL, graph processing, and Machine Learning, all within the same framework.
- *Streaming*: Spark streaming allows large-scale, fault-tolerant streaming computations, integrating batch, interactive, and streaming processes.
- *SparkSQL*: offers full support for HiveQL and UDFs, providing SQL-based querying for Spark data.
- *Performance*: Spark is faster than Hadoop, especially when data is stored in memory, and supports large-scale data storage systems.

Resilient Distributed Datasets are the core data structure in Spark, representing a distributed collection of data that is fault-tolerant and supports parallel operations. Resilient Distributed Datasets allow for operations like map, filter, and join, which are lazily evaluated (executed only when an action is called):

- *Transformations*: operations like map and filter create new Resilient Distributed Datasets but are not executed immediately.
- *Actions*: operations like count and collect trigger execution and return results.

DataFrames are immutable collections of data with named columns built on top of Resilient Distributed Datasets. They offer a user-friendly API, uniform functionality across languages, and performance optimizations.

3.3.2.1 Parquet

Apache Parquet is a columnar storage format optimized for high compression and scan efficiency, commonly used in the Hadoop and Spark ecosystems. It is ideal for storing nested data and is supported by most data processing frameworks. The columnar format uses definition and repetition levels to store data, which is highly efficient for analytic workloads.

Optimized for analytical queries by allowing fast scans of specific columns. Supports high compression ratios, making it ideal for storing large datasets efficiently. Parquet works across multiple languages and data processing frameworks, including Apache Spark and Hadoop.