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 (declarative vs. imperative), 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

Effectively leveraging big data requires the establishment of a comprehensive data management process that encompasses all stages of the data pipeline. This process includes data collection, ingestion, analysis, and the ultimate creation of value. Each stage is critical to transforming raw data into actionable insights that can drive decision-making and generate tangible benefits. The key components of this process are outlined below:

1. *Data collection*: gathering data from a wide range of sources is the foundation of any big data initiative.

2. *Data analysis*: the collected data must be meticulously analyzed to uncover patterns, trends, and insights. This analysis is tailored to the needs of various stakeholders. Analytical methods include descriptive analysis, which provides a snapshot of current data trends, and predictive analysis, which forecasts future developments.

3. *Value creation*: the final step in the data pipeline is the creation of value from the analyzed data. This value can manifest in several ways.

Big data is becoming increasingly prevalent due to several key factors:

- *Declining storage costs*: as hard drives and storage technologies become more affordable, organizations can store vast amounts of data more economically, making it feasible to accumulate and analyze large datasets regularly.

- *Ubiquitous data generation*: in today's digital age, we are all constant producers of data, whether through our interactions on social media, the use of smart devices, or routine activities online. This continuous data generation contributes to the exponential growth of big data.

- *Rapid data growth*: the volume of data is expanding at a rate that far outpaces the growth in IT spending. This disparity drives the need for more efficient and scalable data management solutions to keep up with the increasing data demands across various industries.

Big data is characterized by several key attributes that distinguish it from traditional data management paradigms. These attributes include:

- *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 (such as databases), unstructured data (like text and emails), and multimedia content (including images, videos, and audio).
- *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.

1.1.1 Data analysis

As the amount of data continues to grow, our methods for solving data-related problems must also evolve. In the traditional approach, analysis was typically performed on a small subset of information due to limitations in data processing capabilities. However, with big data, we adopt an innovative approach that allows us to analyze all available information, providing a more comprehensive understanding and deeper insights.

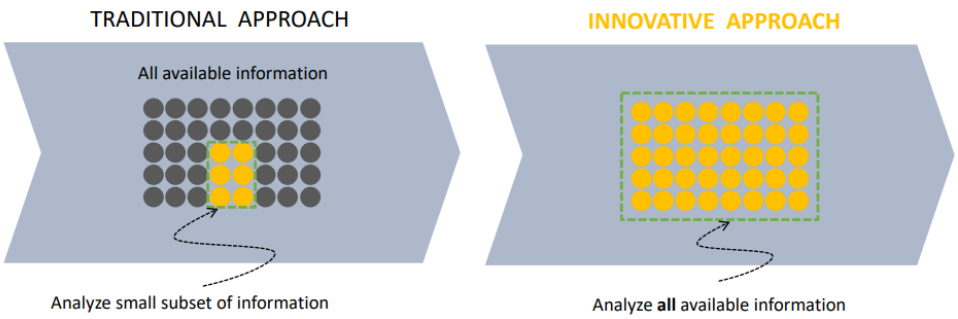

Figure 1.1: More data analyzed

In the traditional approach, we typically start with a hypothesis and test it against a selected subset of data. This method is limited by the scope and size of the data sample. In contrast, the innovative approach used in big data allows us to explore all available data, enabling the identification of correlations and patterns without pre-established hypotheses. This data-driven exploration opens up new possibilities for discovering insights that might have been overlooked using traditional methods.

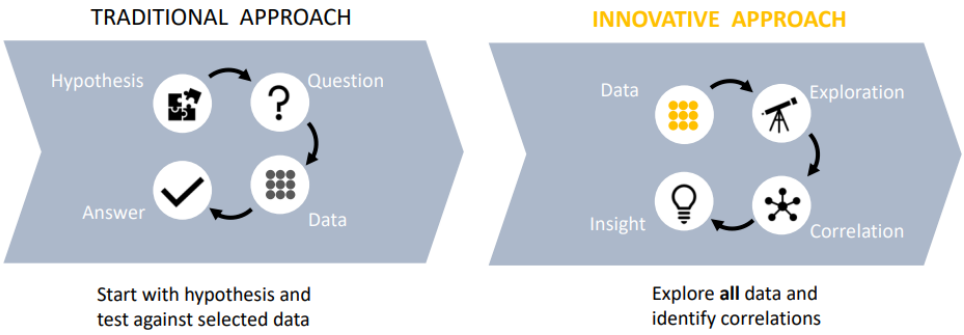

Figure 1.2: Data driven exploration

In the traditional approach, we meticulously cleanse data before any analysis, resulting in a small, well-organized dataset. In contrast, the innovative approach involves analyzing data in its raw form and cleansing it as necessary, allowing us to work with a larger volume of messy information.

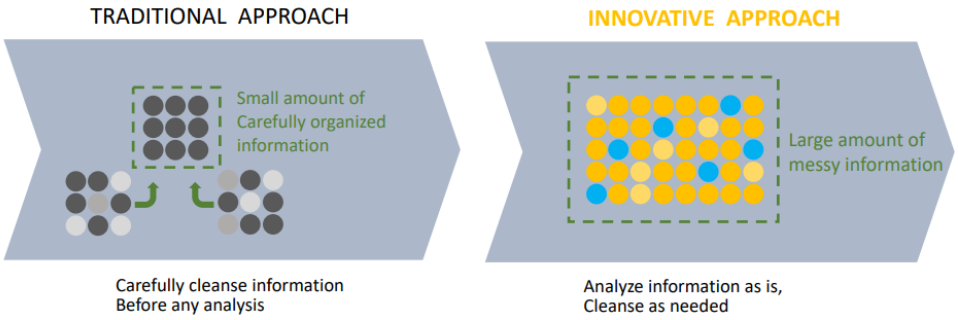

Figure 1.3: Less effort

In the traditional approach, data is analyzed only after it has been processed and stored in a warehouse or data mart. Meanwhile, the innovative approach focuses on analyzing data in motion, in real-time as it is generated.

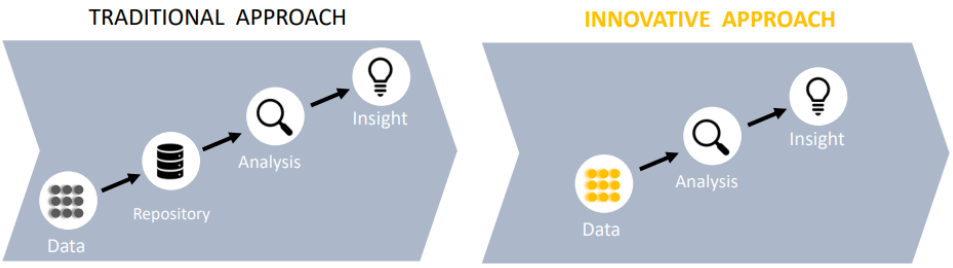

Figure 1.4: Streaming data

1.2 Relational databases

The design levels of a database are the following:

- *Conceptual database design*: constructing an information model, independent from all physical consideration for an enterprise. In entity relationship databases we have: entities, relationships, attributes, attribute domain, and key attributes.

- *Logical database design*: building an organization database based on a specific data model
- *Physical database design*: implementing a database using specific data storage structure(s) and access methods,

1.2.1 Model characteristics

In entity-relationship database models, key components include entities and relationships. An entity represents a distinct real-world object that can be differentiated from others, characterized by a set of attributes. These entities are grouped into an entity set, which consists of similar entities that share the same attributes. Each entity set is identified by a unique key made up of a set of attributes, and each attribute has a defined domain.

Relationships are associations among two or more entities. A relationship set is a collection of similar relationships, where an n-ary relationship set R connects n entity sets $E_1, \ldots, E_n$. Each relationship in this set involves entities from the corresponding entity sets. Notably, the same entity set can participate in different relationship sets or assume various roles within the same set. Additionally, relationship sets can have descriptive attributes. Uniquely, a relationship is defined by the participating entities without relying on descriptive attributes, while the cardinality indicates the number of potential connections between the entities. ISA hierarchies can further enhance the model by adding descriptive attributes specific to subclasses.

Aggregation comes into play when modeling a relationship that involves both entity sets and a relationship set. This technique allows us to treat a relationship set as an entity set, facilitating its participation in other relationships.

Conceptual design Crucial design choices involve determining whether a concept should be modeled as an entity or an attribute, and deciding if it should be represented as an entity or a relationship. It is essential to identify the nature of relationships, considering whether they are binary or ternary, and whether aggregation is appropriate. In the ER model, it is important to capture a significant amount of data semantics. However, some constraints cannot be represented within ER diagrams.

1.3 Relational databases

The SQL standard was first proposed by E. F. Codd in 1970 and became available in commercial DBMSs in 1981. It is based on a variant of the mathematical notion of a relation. Relations are naturally represented by means of tables.

Given n sets $D_1, D_2, \ldots, D_n$, which are not necessarily distinct:

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

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

Definition (*Relation domains*)**.** The sets $D_1, D_2, \ldots, D_n$ are called the domains of the relation.

Definition (*Relation degree*)**.** The number n is referred to as the degree of the relation.

Definition (*Cardinality*)**.** The number of n-tuples is called the cardinality of the relation.

In practice, cardinality is always finite.

Definition (*Ordered set*)**.** A mathematical relation is a set of ordered n-tuples $(d_1, d_2, \ldots, d_n)$ such that $d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n$, where:

- There is no specific ordering between n-tuples.
- The n-tuples are distinct from one another.

The n-tuples are ordered internally: the i-th value comes from the i-th domain.

Example:
Consider a simple mathematical relation:

$$\text{game} \subseteq \text{string} \times \text{string} \times \text{integer} \times \text{integer}$$

For instance:

Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	1	2
Roma	Milan	0	1

Each of the domains has two roles, which are distinguished by their position. The structure is positional.

We can move towards a non-positional structure by associating a unique name (attribute) with each domain, which describes the role of the domain. For instance:

Home team	**Visiting team**	**Home goals**	**Visitor goals**
Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	1	2
Roma	Milan	0	1

Definition (*Relation schema*)**.** A relation schema consists of a name (of the relation) R with a set of attributes $A_1, \ldots, A_n$:

$$R(A_1, \ldots, A_n)$$

Definition (*Database schema*)**.** A database schema is a set of relation schemas with different names:

$$R = \{R_1(X_1), \ldots, R_n(X_n)\}$$

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

Definition (*Instance of a database*)**.** A database instance on a schema $R = \{R_1(X_1), \ldots, R_n(X_n)\}$ is a set of relations $r = \{r_1, \ldots, r_n\}$, where r_i is a relation on R_i.

The relational model imposes a rigid structure on data:

- Information is represented by tuples.
- Tuples must conform to relation schemas.

There are at least three types of null values:

- *Unknown value*: there is a domain value, but it is not known.
- *Non-existent value*: the attribute is not applicable for the tuple.
- *No-information value*: we don"'t know whether a value exists or not (logical disjunction of the above two).

DBMSs typically do not distinguish between these types of nulls and implicitly adopt the no-information value.

An integrity constraint is a property that must be satisfied by all meaningful database instances. It can be seen as a predicate: a database instance is legal if it satisfies all integrity constraints. Types of constraints include:

- Intrarelational constraints (e.g., domain constraints, tuple constraints).
- Interrelational constraints.

Definition (*Key*)**.** A key is a set of attributes that uniquely identifies tuples in a relation.

A set of attributes K is a superkey for a relation r if r does not contain two distinct tuples t_1 and t_2 such that $t_1[K] = t_2[K]$. K is a key for r if K is a minimal superkey (i.e., there exists no other superkey K' of r that is a proper subset of K).

Primary Keys The presence of nulls in keys must be limited. A practical solution is to select a primary key for each relation, on which nulls are not allowed. Primary key attributes are underlined in notation. References between relations are realized through primary keys.

Foreign Keys Data in different relations are correlated by means of values of (primary) keys. Referential integrity constraints are imposed to ensure that these values correspond to actual values in the referenced relation.

1.3.1 ER to relational transformation

To transform an Entity-Relationship (ER) diagram into a relational database schema, the following steps should be performed:

1. *Create a separate table for each entity*:
    - Each attribute of the entity becomes a column in the corresponding relational table.
    - Each instance of the entity set becomes a row in the relational table.
2. *Handle relationships*:
    - For each relationship in the ER diagram, decide whether to represent it as a separate table or as a foreign key in an existing table.
    - Binary relationships with a one-to-many or many-to-one cardinality can often be handled by adding a foreign key to the table corresponding to the many side.
    - Many-to-many relationships typically require the creation of a separate relationship table, where foreign keys from the related entities form the primary key of the new table.

1.4 Data architectures

The data schema ensures typing, coherence, and uniformity within a system.

Definition (*Transaction*)**.** A transaction is an elementary unit of work performed by an application.

Each transaction is encapsulated between two commands: `BEGIN TRANSACTION` and `END TRANSACTION`. During a transaction, exactly one of the following commands is executed:

- `COMMIT WORK` (commit): confirms the successful completion of the transaction.
- `ROLLBACK WORK` (abort): reverts the system to its state before the transaction began.

Definition (*OnLine Transaction Processing*)**.** A transactional system (OLTP) is a system that defines and executes transactions on behalf of multiple, concurrent applications.

1.4.1 Data partitioning

The main goal of data partitioning is to achieve scalability and distribution. Partitioning divides the data in a database and allocates different pieces to various storage nodes. This can be done in two ways:

- *Horizontal partitioning* (sharding): data is divided by rows, where different rows are stored on separate nodes. Sharding is often used to distribute data in large-scale systems, spreading the load across multiple machines.
- *Vertical partitioning*: data is divided by columns, where different columns are stored on different nodes. This method is useful when certain columns are accessed more frequently than others, allowing for optimization of data retrieval.

Partitioning has its advantages and disadvantages. On the plus side, it allows for faster data writes and reads, and comes with low memory overhead. However, it can also lead to potential data loss if not properly managed, especially in cases of node failures or partition mismanagement.

1.4.2 Data replication

The aim of data replication is to provide fault-tolerance and reliable backups. In replication, the entire database is copied across all nodes within a distributed system, ensuring that there are multiple copies available in case of failure.

Replication offers certain benefits. For instance, it provides faster data reads since multiple copies of the data are stored on different nodes, and it greatly increases the reliability of the system, as the risk of losing all copies of the data is significantly reduced.

However, replication also comes with certain drawbacks. It leads to high network overhead, as nodes must constantly synchronize data to ensure consistency. Additionally, replication increases memory overhead since the full dataset is duplicated across all nodes in the system.

1.4.3 Scalability

We aim to create a system with elasticity.

Definition (*Elasticity*)**.** Elasticity refers to the ability of a system to automatically scale resources up or down based on demand, ensuring efficient use of resources and cost-effectiveness without compromising performance.

Data Ingestion Data ingestion is the process of importing, transferring, and loading data for storage and future use. It involves loading data from a variety of sources and may require altering or modifying individual files to fit into a format that optimizes storage efficiency.

Data Wrangling Data wrangling is the process of cleansing and transforming raw data into a format that can be analyzed to generate actionable insights. This process includes understanding, cleansing, augmenting, and shaping the data. The result is data in its optimal format for analysis.

1.5 NoSQL Databases

NoSQL databases are designed to offer greater flexibility and scalability, making them well-suited for dynamic data structures in modern applications. Unlike traditional relational databases that rely on fixed schemas, NoSQL databases often operate without an explicit schema, or they use flexible schemas that can evolve over time. This adaptability allows them to accommodate various types of data, including unstructured and semi-structured formats such as JSON, XML, or key-value pairs.

The lack of a rigid schema enables NoSQL databases to manage large-scale, constantly changing datasets efficiently. This characteristic makes them ideal for applications where data formats are unpredictable or subject to frequent changes, such as social media platforms, IoT systems, and real-time analytics.

Paradigmatic shift The rise of Big Data has led to a fundamental shift in how databases are designed and used. Traditional databases typically follow a schema on write approach, where a well-defined schema must be agreed upon before data can be stored. This model is limiting in fast-changing environments where the data structure may not be fully known at the time of ingestion, resulting in the potential loss of valuable information. NoSQL databases adopt a schema on read approach, allowing data to be ingested without predefined structure. The minimal schema necessary for analysis is applied only when the data is read or queried. This flexibility allows for more comprehensive data retention and analysis, enabling new types of queries and insights to be derived as the requirements evolve.

Object-Relational Mapping In traditional databases, Object-Relational Mapping (ORM) is used to bridge the gap between object-oriented programming languages and relational databases, a problem known as the impedance mismatch. Despite the existence of ORM solutions, this process is often complex and can hinder performance and flexibility. NoSQL databases, particularly object-oriented and document-based databases, can eliminate or reduce the impedance mismatch by storing data in formats that align more naturally with the objects in application code. While early object-oriented database systems were commercially unsuccessful, modern NoSQL systems provide a more pragmatic solution to these challenges.

Data lake NoSQL databases often serve as the backbone of data lakes, where raw, unstructured, and structured data is stored in its native format. Data lakes are designed to allow for future analysis, without requiring immediate transformation into a rigid schema, making them highly compatible with NoSQL databases.

Scalability Traditional SQL databases scale vertically, meaning performance improvements come from upgrading to more powerful hardware with better memory, processing power, or storage capacity. However, vertical scaling has physical and financial limits, and adding data to a traditional SQL system can degrade its performance over time. In contrast, NoSQL databases are designed to scale horizontally. This means that when the system needs more capacity, additional machines (nodes) can be added to the cluster, allowing the database to distribute both data and computational load across multiple nodes. This architecture is especially effective for handling the vast datasets and high-throughput demands characteristic of Big Data applications.

1.5.1 CAP theorem

The CAP theorem highlights the trade-offs inherent in distributed systems.

Theorem 1.5.1. *A distributed system cannot simultaneously guarantee all three of the following properties:*

- *Consistency: all nodes see the same data at the same time.*
- *Availability: every request receives a response, whether it is successful or not.*
- *Partition tolerance: the system continues to operate even if communication between nodes is interrupted due to network failures.*

NoSQL databases typically sacrifice either consistency or availability, depending on the specific use case. Systems can be categorized as:

- *CP* (Consistency, Partition tolerance): prioritize data correctness at the cost of availability during network failures.
- *AP* (Availability, Partition tolerance): prioritize availability, potentially returning stale or outdated data during partition events.

Understanding this trade-off is crucial for designing systems that balance performance, reliability, and scalability based on specific application requirements.

BASE properties While traditional databases follow the ACID (Atomicity, Consistency, Isolation, Durability) principles, many NoSQL databases adhere to the BASE model:

- *Basically Available*: the system guarantees availability, even if data is not fully consistent.
- *Soft state*: the state of the system may change over time, even without input (due to eventual consistency).
- *Eventual consistency*: The system will eventually become consistent, but intermediate states may be inconsistent.

This model is particularly useful in environments where high availability and scalability are prioritized over strict consistency, such as in distributed systems that can tolerate temporary inconsistencies.

1.5.2 NoSQL history

NoSQL databases have a rich history, beginning as early as the 1960s. xKey milestones include:

- 1965: multiValue databases developed by TRW.
- 1979: AT&T releases DBM, an early precursor to NoSQL systems.
- 2000s: modern NoSQL databases emerge, including Google BigTable (2004), CouchDB (2005), Amazon Dynamo (2007), and MongoDB (2009).
- 2009: the term NoSQL is reintroduced to describe a new generation of non-relational databases optimized for scalability and flexibility.

1.5.3 NoSQL taxonomy

NoSQL databases can be categorized into several types:

- *Key-value stores*: data is stored as key-value pairs. Examples include Redis and Azure Table Storage.
- *Column stores*: data is stored in columns, making them highly efficient for analytical queries. Examples include Cassandra and Hadoop.
- *Document stores*: these databases store data as documents, often in formats like JSON or BSON. Examples include MongoDB and CouchDB.
- *Graph databases*: these databases represent data in terms of nodes and relationships (edges), ideal for complex relationship mapping. An example is Neo4j.

Each type of NoSQL database has its strengths and is designed to meet different kinds of scalability, flexibility, and performance needs.

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 (JVM), which also hosts APIs that allow users to interact with the database.

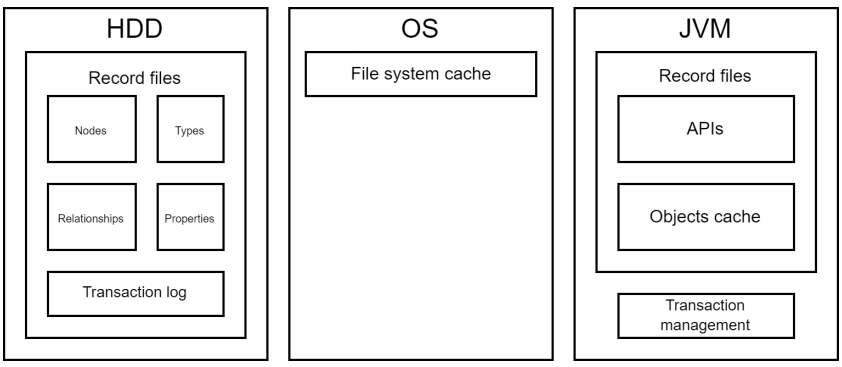

Figure 2.1: Neo4j architecture

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* (edges): 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.

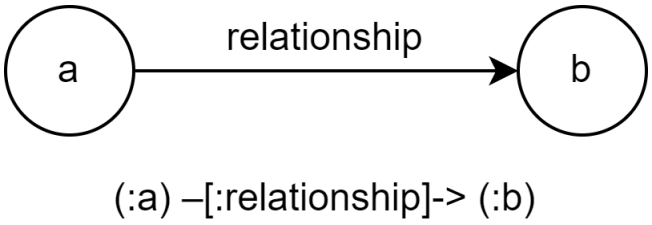

Figure 2.2: Neo4j data model

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 database

In traditional relational databases, data is distributed across multiple tables. However, in business applications, it is often beneficial to structure the data into a single, cohesive document that pulls together relevant information from various sources. This approach provides a more intuitive representation of the data, simplifying query processes by avoiding the need to join multiple tables. Additionally, document-oriented databases are highly flexible, allowing for the easy addition of attributes, which makes handling schema changes much more straightforward. The document model also closely aligns with object-oriented programming paradigms, effectively resolving the impedance mismatch problem that arises when trying to map objects to relational tables.

2.2.1 MongoDB

MongoDB is a widely-used, open-source, document-oriented database. It stores data in flexible, JSON-like documents, offering developers agility and scalability. With MongoDB, the schema is dynamic, allowing for flexible and evolving data models. Furthermore, it supports automatic data sharding, enabling seamless horizontal scaling. Key advantages of MongoDB include:

- *General-purpose*: MongoDB offers a rich data model, full-featured indexes, and a sophisticated query language that can handle a wide variety of use cases.
- *Ease of use*: its structure allows for an easy mapping to object-oriented code, with native drivers for popular programming languages. The setup and management process is simple and developer-friendly.
- *Performance and scalability*: MongoDB operates at in-memory speed whenever possible, and its built-in auto-sharding ensures smooth scaling without downtime. Developers can dynamically add or remove capacity as needed.

Security features SSL encryption between client and server, and intra-cluster communication. Fine-grained authorization controls at the database level, supporting read-only, read and write, and administrative roles.

MongoDB processes MongoDB operates with three core processes:

- *Mongod*: this is the primary process that runs the MongoDB database instance. It handles all the database operations, data storage, and query processing.
-
- *Mongos*: Responsible for managing the sharding architecture, mongos acts as a query router. It directs client requests to the appropriate mongod instances based on the sharding configuration. You can deploy either a single mongos for the entire system or multiple mongos instances (e.g., one per client) to reduce network latency and improve performance.
- *Mongo*: this is an interactive command-line shell used by clients to interact with MongoDB, perform queries, administrative tasks, and more.

Sharding Sharding is MongoDB's method for partitioning large datasets across multiple servers to achieve horizontal scaling, optimize performance, and provide resilience. Key features of MongoDB sharding are:

- *Scale*: Designed to handle the massive workloads of modern applications, ensuring scalability as data volume grows.
- *Geo-locality*: Allows geographically distributed deployments, optimizing user experience across different regions.
- *Hardware optimization*: Enables fine-tuning performance versus cost by distributing data intelligently across available resources.
- *Lower recovery times*: Facilitates faster recovery during failures, supporting stringent Recovery Time Objectives (RTO).

A shard key is defined by the data modeler and determines how MongoDB partitions data across shards. The data is divided into chunks based on this key, which are then evenly distributed across the available shards on different physical servers. The sharding unfolds in the following steps:

- Initially, the system starts with a single chunk, but as the dataset grows, MongoDB automatically splits and migrates chunks to ensure balanced distribution across shards. The default maximum chunk size is 64MB.
- MongoDB routes queries directly to the appropriate shard(s), reducing query overhead.
- A config server stores information about the shard ranges and their locations. In production environments, it is recommended to have three config servers to ensure high availability.

The sharding can be:

- *Range*: data is partitioned based on a continuous range of shard keys (e.g., {deviceId}). Composite keys are also supported (e.g., {deviceId, timestamp}).

- *Hash*: MongoDB applies an MD5 hash to the shard key, ensuring that data is randomly distributed across the available shards. This technique minimizes hotspots and ensures even distribution.
- *Tag*: this method allows specific shards to be tagged with labels, ensuring that certain data subsets (e.g., users from a specific region) are stored on a designated group of shards. This is particularly useful for 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 focuses on consistency and partition tolerance.

2.2.2 Data model

MongoDB stores data in JSON format, which is ideal for web applications due to its readability and flexibility. JSON structures can be easily adapted to various needs, and MongoDB's dynamic schema approach means that new fields can be added without requiring changes to the existing schema. Key features of the MongoDB data model:

- MongoDB keeps frequently accessed data in memory to optimize performance.
- It is highly horizontally scalable, allowing servers to be added as needed.
- MongoDB organizes data in contiguous regions for better locality and access speed.

Although MongoDB lacks traditional database features such as schemas, transactions, and joins, it is highly optimized for modern application development. Large documents can be stored using GridFS, and the maximum document size is 16MB.

Binary JSON MongoDB uses BSON, a binary representation of JSON documents, to improve speed and efficiency. BSON supports more data types than JSON, including date and byte array types, while optimizing space and serialization speed. Each document must have a unique identifier, which MongoDB can automatically generate if not provided. MongoDB also allows for embedding multiple documents within a single document, which simplifies data retrieval and reduces the need for complex joins.

2.2.3 Query language

MongoDB allow to perform the CRUD operations:

- *Create*: we can create an entire database:

```
use database_name
```

We may create a collection:

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

And also inser a new document:

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

- *Read*: we can query for all elements:

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

With the `find` function we can insert some filters.

We can also aggregate documens, by pipelining documents from a collection through an aggregation pipeline. Expressions produce output documents based on calculations performed on input documents:

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

- *Update*: the general command to update documents is:

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

We can also use the save method to replace an already existsing document:

```
db.students.save()
```

- *Delete*: we can drop a database:

```
db.dropDatabase()
```

We can drop a collection or a document:

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

2.3 Key-value database

Key-value databases store data as a collection of key-value pairs, where each key acts as a unique identifier that points to a corresponding value. This structure allows for efficient retrieval of data, making key-value databases particularly useful in scenarios that require fast lookups by key. 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.

Key-value databases are often used as the foundation for applications requiring high performance in terms of speed and scalability, and they form the backbone for search operations by id.

2.3.1 Redis

Key features of Redis include:

- *Advanced data structures*: Redis values can be more than just simple strings or numbers—they can represent data structures like lists, sets, or even geospatial indexes.
- *Atomic operations*: 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.
- *Versatility*: 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.

Redis follows a unique path in the evolution of key-value databases, as it directly exposes complex data types as part of its interface, without adding extra abstraction layers. This makes Redis particularly well-suited for use cases where performance and simplicity are critical.

While Redis is not a direct replacement for relational databases or document stores, it complements them well. Redis can be used alongside SQL databases for fast access to frequently queried data, or alongside NoSQL databases to provide rapid access to specific data sets.

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.

Advantages 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 ANSI 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.

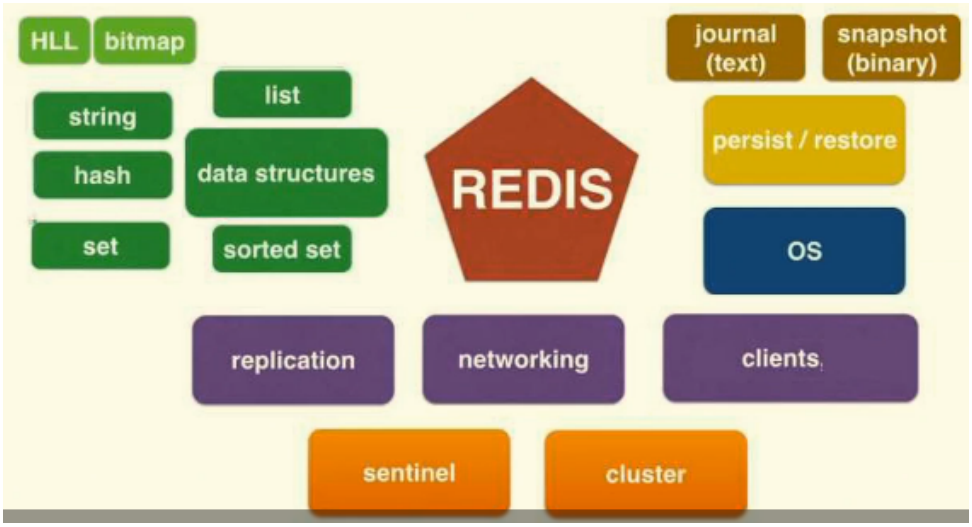

Figure 2.3: Redis architecture

Redis offers two persistence mechanisms:

- *Redis Database Snapshots* (RDB): captures a snapshot of the dataset at specified intervals.
- *Append-Only File* (AOF): 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.

Topologies 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_field string_value
GET string_field
// set or increment numbers values
SET (int)string_field 1
INCRBY (int)string_field 1
// get multiple keys at once
MGET string_field (int)string_field
// set multiple keys at once
MSET string_field string_value (int)string_field 1223
// get the length of a string
STRLEN string_field
// update a value retrieving the old one
GETSET string_field 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
// refield a key
REfield bar new_bar
// 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
RINDEX key_value
LINDEX key_value
// get a range of elements
RRANGE key_value 0-1
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

Cache A cache is a collection of stored data duplicates, designed to quickly provide values that are either difficult or time-consuming to retrieve or compute. Caching enhances performance by making frequently requested data readily available, saving time compared to re-fetching or recalculating. Caches use a simple key-value storage model, typically involving operations to save, retrieve, and delete values. Cache systems often incorporate replacement policies to manage limited storage space efficiently. A cold cache holds no stored data, while a warm cache has useful data loaded, resulting in higher cache hits. The effectiveness of caching depends on the balance between cache hits and misses, with a high hit ratio indicating efficient performance.

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.

Optimization is key to maximizing Memcached's benefits. 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 column-oriented database, or columnar database, stores data in columns rather than rows. This approach is optimized for Online Analytical Processing (OLAP) 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, making them highly efficient for read-heavy workloads. 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 often trade I/O for CPU by leveraging 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 and its column-oriented approach make it particularly suited for high-availability, large-scale, write-intensive workloads, where distributed, flexible data storage is essential.

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.

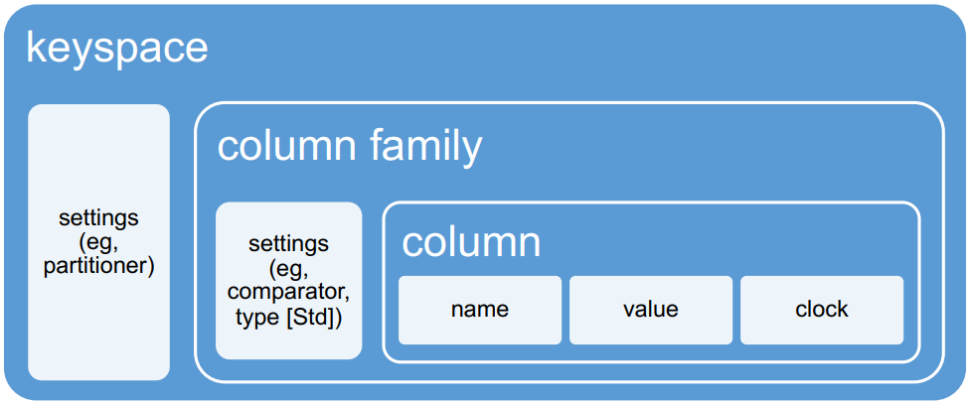

Figure 2.4: Cassandra data model

2.4.1.3 Query language

Cassandra supports a unique query mechanism based on a slice predicate, allowing precise control over returned columns: SliceRange specifies start and end column names, direction (reverse), and count (similar to SQL LIMIT).

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<ColumnOrSuperColumn>
// retrieve slices for multiple keys based on a SlicePredicate
multiget_slice() : Map<key, List<ColumnOrSuperColumn>>
// 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()
```

SQL Cassandra also supports SQL.

APPENDIX A

Additional topics

A.1 Graph theory

In mathematics, a graph is a structure composed of nodes (or vertices) connected by edges (or lines). Graph theory studies these structures and their properties.

Definition (*Graph*)**.** A graph G is an ordered triple $G = (V, E, f)$, where:

- V is a set of vertices (or nodes),
- E is a set of edges, each representing a connection between two vertices,
- f is a function that maps each edge in E to an unordered pair of vertices in V.

Definition (*Vertex*)**.** A vertex is a fundamental element in a graph, represented visually as a point or dot. The vertex set of a graph G is usually denoted $V(G)$ or V.

Definition (*Edge*)**.** An edge is a set of two vertices, often depicted as a line connecting them. The set of all edges in G is denoted $E(G)$ or E.

Definition (*Simple graph*)**.** A simple graph is a graph without multiple edges (no repeated edges) and without loops (edges that connect a vertex to itself).

Definition (*Path*)**.** A path in a graph is a sequence of vertices in which each adjacent pair is connected by an edge.

Definition (*Simple path*)**.** A path is considered simple if all vertices in the path are distinct.

Definition (*Cycle*)**.** A cycle is a path that starts and ends at the same vertex.

Definition (*Cyclic graph*)**.** A graph is cyclic if it contains at least one cycle.

Definition (*Connected graph*)**.** A graph is connected if there exists a path between any pair of vertices, allowing traversal between any two vertices in the graph.

Definition (*Strongly connected graph*)**.** A directed graph is strongly connected if there is a directed path from any vertex to every other vertex.

Definition (*Sparse graph*)**.** A sparse graph is one in which the number of edges is close to the number of vertices:

$$|E| \approx |V|$$

Definition (*Dense graph*)**.** A dense graph is one in which the number of edges is close to the square of the number of vertices:

$$|E| \approx |V|^2$$

Definition (*Weighted graph*)**.** A weighted graph assigns a weight to each edge, typically represented by a weight function $w : E \to \mathbb{R}$.

Definition (*Directed graph*)**.** A directed graph, or digraph, is a graph in which each edge has a direction, meaning edges are ordered pairs of vertices.

Definition (*Bipartite graph*)**.** A graph is bipartite if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that every edge connects a vertex in V_1 to a vertex in V_2.

Definition (*Complete graph*)**.** A complete graph, denoted K_n, is a graph in which every pair of vertices is connected by an edge. A complete graph with n vertices has $\frac{n(n-1)}{2}$ edges.

Definition (*Planar graph*)**.** A planar graph can be drawn on a plane without any edges crossing. The complete graph K_4 is the largest complete planar graph.

Definition (*Tree*)**.** A tree is a connected, acyclic graph. In a tree, there is exactly one path between any pair of vertices.

Definition (*Hypergraph*)**.** A hypergraph generalizes a graph by allowing edges (called hyperedges) to connect any number of vertices. Formally, a hypergraph is a pair (X, E), where X is a set of vertices and E is a set of subsets of X, each subset representing a hyperedge.

Definition (*Degree*)**.** The degree of a vertex is the number of edges incident to it.

For directed graphs:

- *In-degree*: the number of edges directed toward the vertex.
- *Out-degree*: the number of edges directed away from the vertex.
- *Degree*: sum of out-degree and in-degree

Definition (*Subgraph*)**.** A subgraph of G is a graph whose vertex set and edge set are subsets of those of G. Conversely, G is called a supergraph of this subgraph.

Definition (*Spanning subgraph*)**.** A spanning subgraph H of G has the same vertex set as G but possibly fewer edges.

A.1.1 Graph data structure

In computer science, a graph is defined as an abstract data type composed of a set of nodes, and a set of edges. These elements establish relationships between the nodes, reflecting the mathematical concept of graphs. Graphs can be represented in several ways, including:

- *Matrix representation*: incidence matrix (edge data in relation to vertices, size $|E| \times |V|$) or adjacency matrix (adjacency or edge weights, size $|V| \times |V|$).
- *List representation*: edge list (pairs of vertices with optional weights and additional data), and adjacency list (collection of lists or arrays where each list corresponds to a vertex and contains its adjacent vertices).