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.

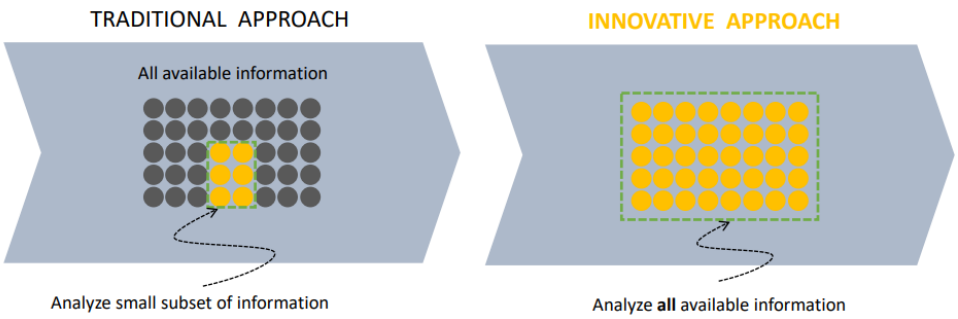

Figure 1.1: More data analyzed

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.

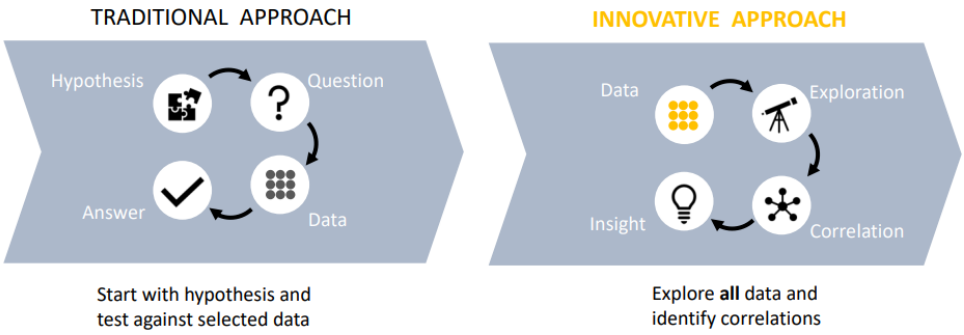

Figure 1.2: Data driven exploration

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.

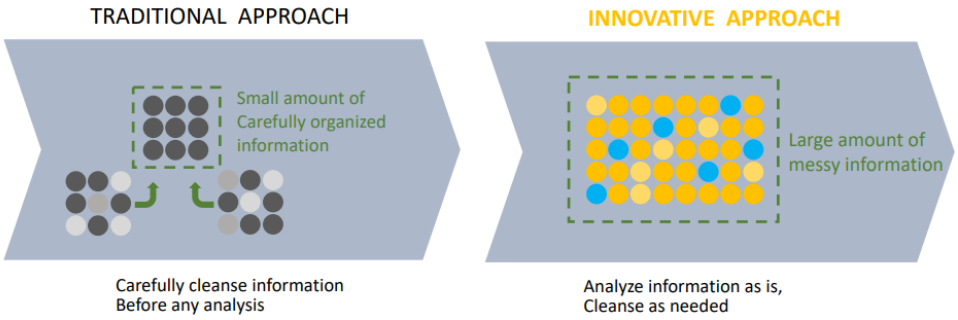

Figure 1.3: Less effort

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.

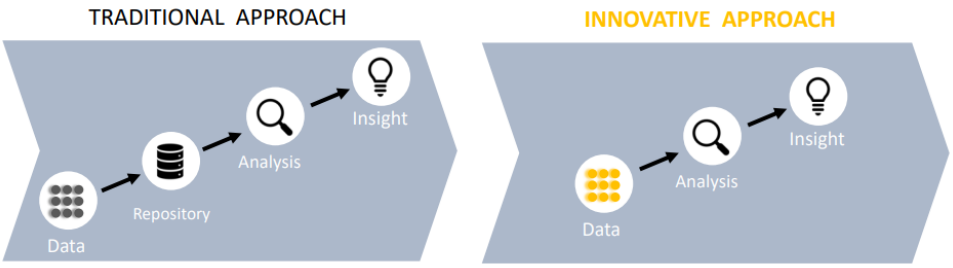

Figure 1.4: Streaming data

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.

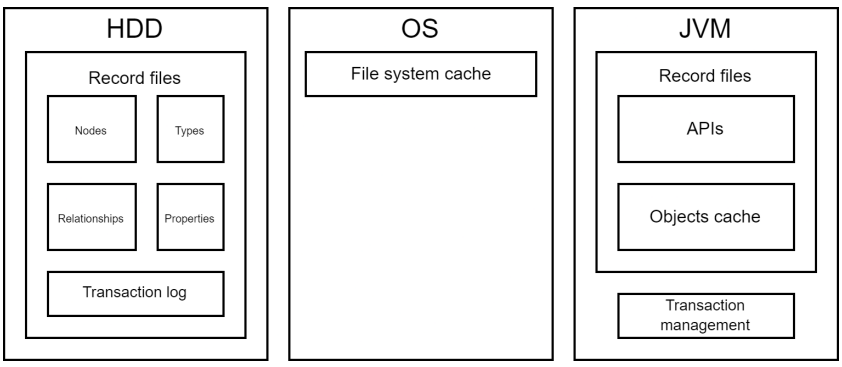

Figure 2.1: Neo4j architecture

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.

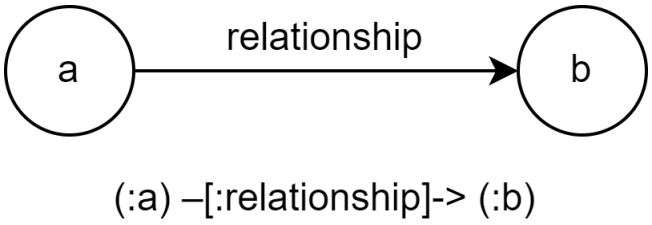

Figure 2.2: Neo4j data model

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 Document-oriented 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 (explained earlier).

- `$sort`: Orders the documents based on specified fields.
- `$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.students.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, 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.

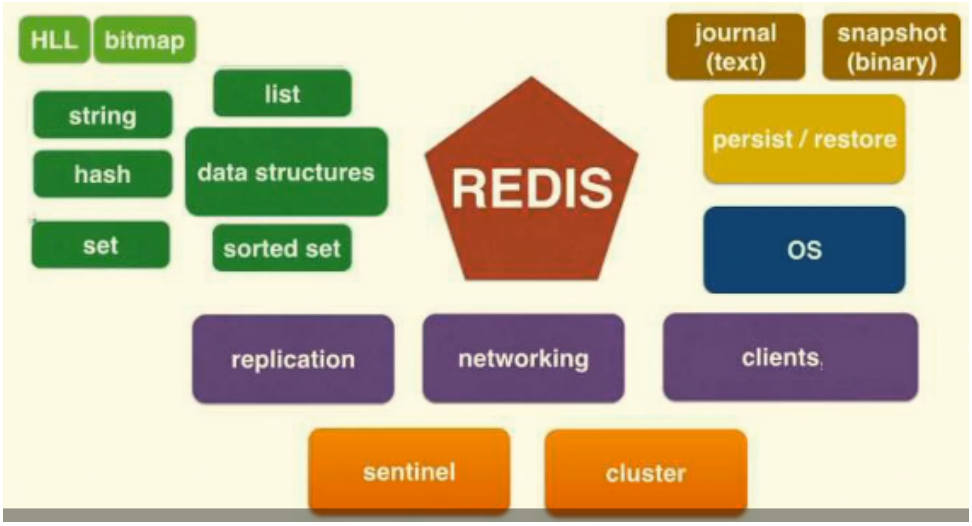

Figure 2.3: Redis architecture

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 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.

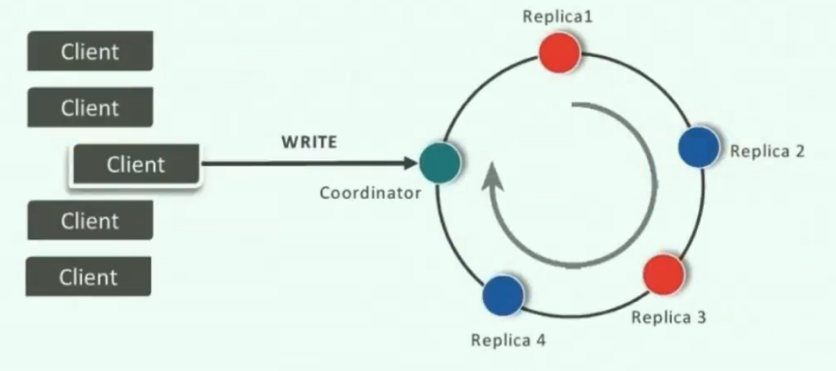

Figure 2.4: Ring architecture

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 Bboom 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.

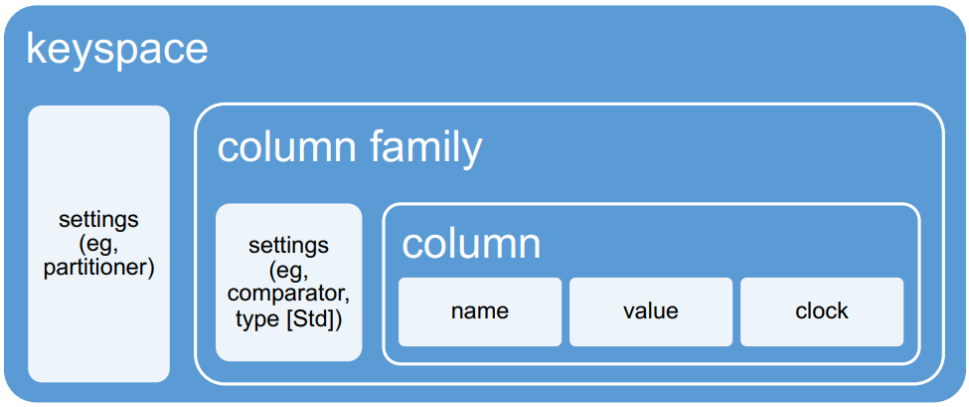

Figure 2.5: Cassandra data model

2.4.1.3 Query language

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 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.

In case we need to use multiple keys for a family of column we may use an aggregate key, that comprises all the key that may be used in the queries:

```
attribute1:attribute2: { user1, user2}
```

Queries can be performed using the first attribute or a combination of attributes, depending on the data model.

Columns within a row are sorted based on the specified comparator, either through `CompareWith` for primary columns or `CompareSubcolumnsWith` for subcolumns. 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.

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.

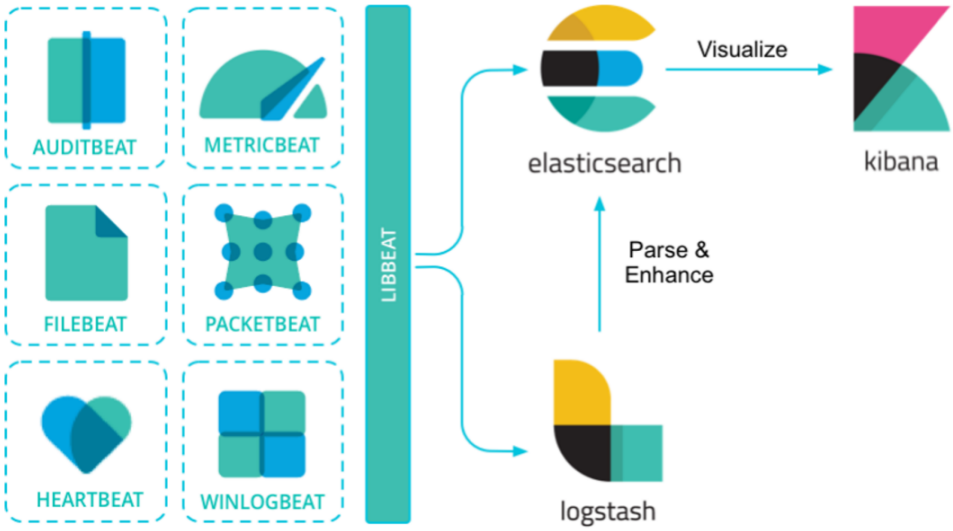

Figure 2.6: ELK stack

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:

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

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:

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

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
}
```

his 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
{
    "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 and Beats

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.

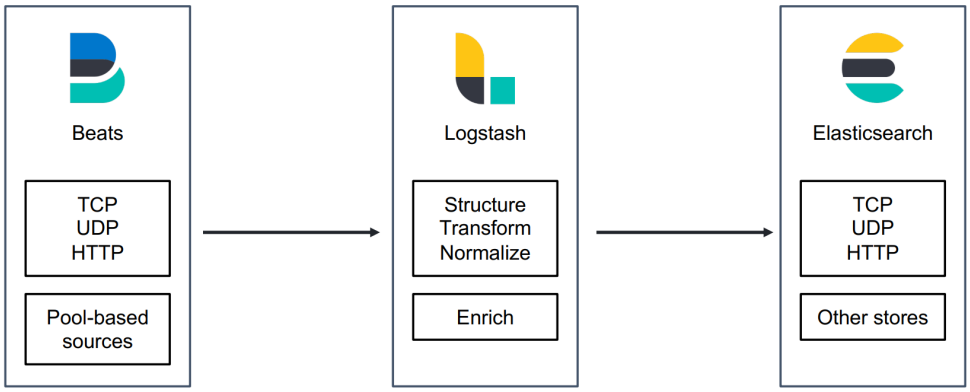

Figure 2.7: Logstash data flow

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 (DLQ) 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. These dashboards can combine basic visualizations with location-based analyses, time series, machine learning insights, and graph/network visualizations, making it an invaluable tool for exploring and understanding complex data.

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).