Systems And Methods For Big And Unstructured Data

Theory

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

1 Introduction **1**
- 1.1 Big data . . . 1
  - 1.1.1 Data analysis . . . 2
- 1.2 Relational databases . . . 3
  - 1.2.1 Model characteristics . . . 4
- 1.3 Relational databases . . . 4
  - 1.3.1 ER to relational transformation . . . 6
- 1.4 Data architectures . . . 7
  - 1.4.1 Data partitioning . . . 7
  - 1.4.2 Data replication . . . 7
  - 1.4.3 Scalability . . . 8
- 1.5 NoSQL Databases . . . 8
  - 1.5.1 CAP theorem . . . 9
  - 1.5.2 NoSQL history . . . 10
  - 1.5.3 NoSQL taxonomy . . . 10

2 NoSQL databases **11**
- 2.1 Graph databases . . . 11
  - 2.1.1 Neo4j . . . 12
  - 2.1.2 Data model . . . 12
  - 2.1.3 Query language . . . 12

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.

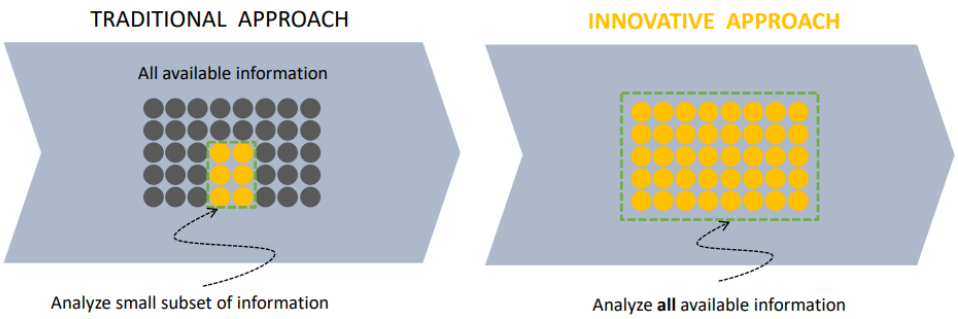

Figure 1.1: More data analyzed

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.

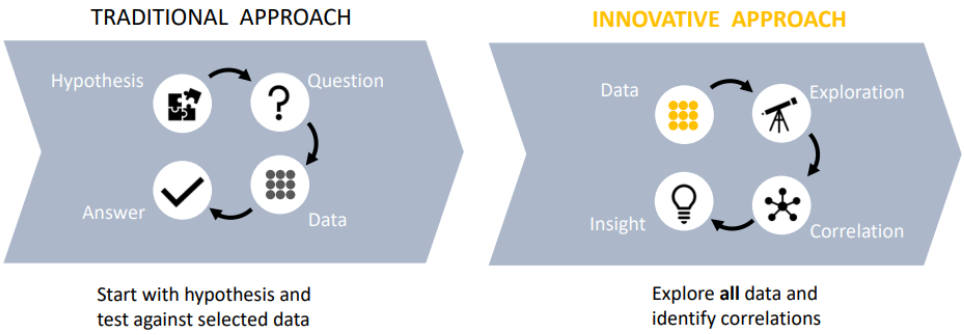

Figure 1.2: Data driven exploration

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.

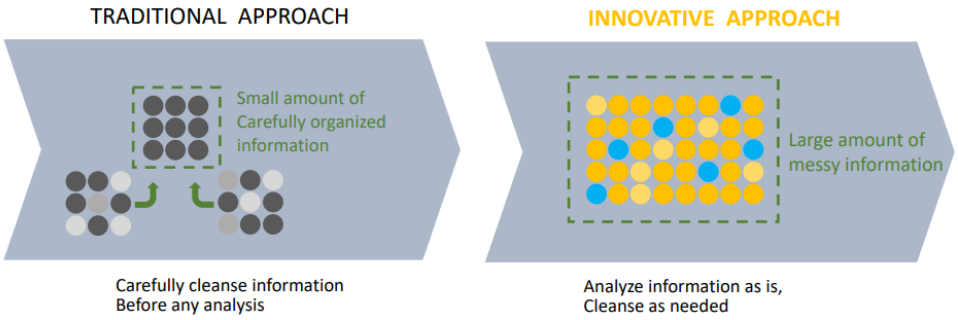

Figure 1.3: Less effort

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.

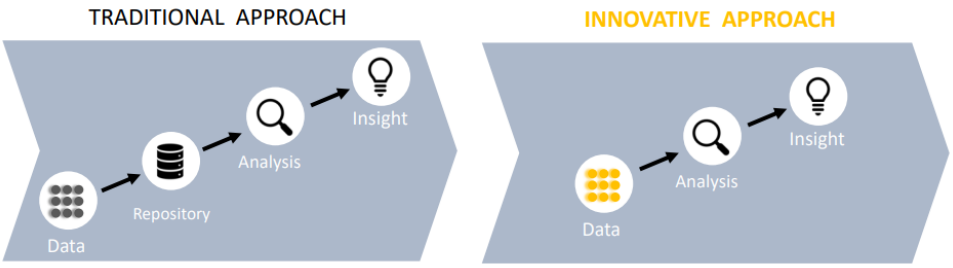

Figure 1.4: Streaming data

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 (data about the entities or relationships). Key features of graph databases include:

- *Index-free adjacency*: each node directly references its adjacent nodes, eliminating the need for costly index lookups during traversal.
- *Relationships as first-class citizens*: in graph databases, edges, or relationships, carry much of the critical information.

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 excel when working with datasets that are rich in relationships, making them an 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, where the relationships between entities are as important as the entities themselves.
- *Natural fit for object-oriented models*: graph databases map more intuitively to object-oriented applications, as 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, such as finding paths or analyzing networks, are much faster compared to relational databases.

One significant challenge with graph databases is that the complexity of the data model can escalate rapidly, leading to maintainability issues. As relationships grow in number and intricacy, managing the graph can become increasingly difficult.

Additionally, performing queries on graph databases can be complex. Crafting efficient queries often requires a deep understanding of the graph structure and the relationships between nodes, which can complicate both development and performance optimization. To simplify the quering process we run a graph matching approach (pattern matching)

2.1.1 Neo4j

Neo4j, developed by Neo Technologies, is the most popular graph database available today. It is implemented in Java and is open-source, making it accessible for various applications. The key features of Neo4j are:

- *Schema-free*: Neo4j allows for a flexible data model where data does not need to conform to a specific schema, facilitating easier updates and modifications.
- *ACID compliance*: it ensures atomicity, consistency, isolation, and durability for logical units of work, which is crucial for maintaining data integrity.
- *User-friendly*: Neo4j is designed to be easy to get started with and use, providing a smooth onboarding experience for new users.
- *Extensive documentation and community*: it boasts thorough documentation and a large, active developer community, making it easier to find support and resources.
- *Multi-language support*: Neo4j supports a wide range of programming languages, including Java, Python, Perl, Scala, and its own query language, Cypher.

Neo4j is primarily intended as an operational database rather than a dedicated analytics platform. It excels in managing relationships and efficiently accessing nodes, although it may not be as effective for comprehensive graph-wide analyses.

2.1.2 Data model

Neo4j is based on:

- *Nodes*: represent entities, equipped with labels (types) and attributes (properties).
- *Edges*: serve as the connections between nodes, providing context and relationships.
- *Indexes*: enhance query performance by allowing quick lookups of nodes and relationships.

2.1.3 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. One of the standout features of Cypher is its emphasis on relationships, which enables users to easily navigate and manipulate complex graph structures. This relationship-centric approach simplifies the querying process compared to traditional SQL, where handling joins can become cumbersome and complex. Many of Cypher's capabilities have been specifically developed to address common challenges faced with SQL, enhancing its usability and efficiency in dealing with graph data.

The Cypher query language in Neo4j supports several key operations for managing and querying graph data. Below is an overview of the primary operations:

- *Data creation*: to create new nodes and relationships in Neo4j, Cypher provides the `CREATE` command.

```
CREATE (n:Label {propertyKey: value, ... })
```

To create a new relationship we write:

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

Note that a node may have two labels:

```
CREATE (n:Label1:Label2 {propertyKey: value, ... })
```

- *Data importing*: we can aslo import an entire graph from a csv file in the following way:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:customers.csv" AS row
CREATE (:Customer {companyName: row.CompanyName, ID: row.CustomerID});
```

The periodic commit is used to guarantee the ACID properties.

- *Data merging*: To ensure you don't create duplicate nodes or relationships, use the merge operation:

```
MERGE (p:Person {name: 'John Doe'})
ON CREATE SET p.age = 30
ON MATCH SET p.lastUpdated = date()
```

We can also merge edges in the same way, avoiding duplicates.

- *Index creation*: create an index over a certain data type:

```
CREATE INDEX ON :Customer(customerID);
```

- *Constraints creation*: create a constraint over a certain data type:

```
CREATE CONSTRAINT ON (c:Customer)
ASSERT c.customerID IS UNIQUE;
```

- *Data querying*: the general query in Cypher is:

```
MATCH (user)-[:FRIEND]-(friend)
WITH user, count(friend) AS friends
ORDER BY friends DESC
SKIP 1 LIMIT 3
RETURN user
```

Aggregation can be used (`COUNT`). `WITH` separates query parts explicitly, to declare the variables for the next part. `SKIP` skips results at the top and `LIMIT` limits the number of results. We can also add an `*` to a relationship to find all the nodes not directly connected.