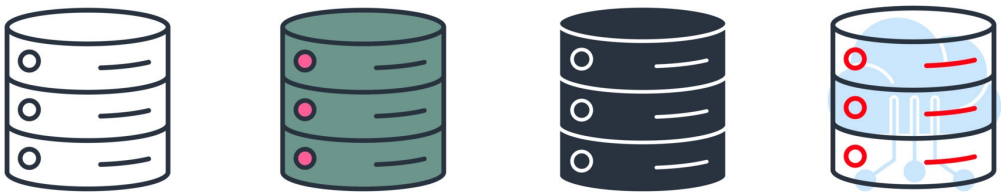
NOTES

Data Management

2223-1-FDS01Q001-FDS01Q001M



Vittorio Haardt
853268
vittoriohaardt@gmail.com

May 22, 2024

Abstract

At the end of the module students will be able to select, design and query a database (relational or not) according to their application needs.

Students will be able to use a NoSql database management system to acquire, memorize and query semi structured data

At the end of the course students will have acquired skills in analysis, evaluation and, to a lesser extent, development of complex and interactive infographics.

Contents

Chapter 1

Data Lifecycle

The data lifecycle is the series of stages that data goes through from the point of its creation or collection to its eventual disposal. The goal of research is the primary driving force behind the data lifecycle. It determines the specific data that is needed and the questions that the data will be used to answer. Finding the data is the next stage in the data lifecycle. This can involve searching for publicly available data sources or collecting data through surveys or experiments. Data preparation is a crucial step in the data lifecycle, as it involves cleaning and organizing the data so that it is ready for analysis. This can be a time-consuming process, as it often involves dealing with missing or incorrect data and formatting the data so that it is consistent and usable. Data exploration is the process of examining the data to understand its characteristics and uncover patterns or trends. This can involve visualizing the data or using statistical techniques to analyze it. Data modeling involves using the data to build predictive models or to test hypotheses about relationships within the data. Presentation and distribution involve sharing the results of the data analysis with others, either through reports, presentations, or by making the data itself available to others.

The data science pipeline refers to the series of steps that a data scientist takes to solve a problem using data.

- **Data acquisition**: This step involves collecting and acquiring the data that will be used to solve the problem. This can involve sourcing data from external sources, such as public data sets or data collected through surveys, or using internal company data.
  - *Data discovery*: This is the process of identifying the data sources that are relevant to the problem being solved and evaluating their suitability for the project.
  - *Data acquisition techniques*: There are various techniques that can be used to acquire data, including web scraping, APIs, database queries, and data dumps. The appropriate technique will depend on the specific data source and the needs of the project.
- **Data storage**: After the data has been cleaned and prepared for analysis, it needs to be stored in a way that makes it accessible and usable for later stages in the pipeline. This can involve storing the data in a database or in a file system.
- **Data preparation**: is an important step in the data science pipeline, as it involves cleaning and organizing the data so that it is ready for analysis.
  - *Data cleaning*: Before the data can be analyzed, it needs to be cleaned to ensure that it is accurate and consistent. This can involve tasks such as handling missing values, dealing with outliers, and ensuring that the data is in a consistent format.

- *Data integration*: This involves combining data from multiple sources and ensuring that it is consistent and coherent. This can involve tasks such as merging data sets, standardizing data formats, and dealing with conflicts or inconsistencies in the data.

Other common stages in the data science pipeline include exploratory data analysis, data modeling, visualization and reporting, and deployment.

Data management refers to the processes and practices used to ensure that data is properly collected, stored, and maintained throughout its lifecycle. In the context of data science, effective data management is essential for ensuring that the data used in a project is accurate, consistent, and available for analysis. The entire data science process can be executed multiple times as new data becomes available or as the business needs change. It is important to have a system in place for managing the data so that it is easily accessible and usable for these repeated processes. The same dataset can be used in different data science processes, and it is important to manage the data in a way that ensures that it is consistent and up-to-date across all of these processes.

One approach to data management is the use of a data lake, which is a centralized repository that allows data to be stored in its raw, unstructured form. This allows the data to be easily accessed and queried by multiple data science processes, while also allowing for the addition of new data as it becomes available. Another approach is master data management, which involves establishing and maintaining a single, consistent version of key data elements, such as customer or product data. This helps to ensure that data is consistent and accurate across all data science processes that use it.

Chapter 2

Data acquisition

2.1 Data acquisition

In a data science pipeline, data acquisition is the process of acquiring and importing data for analysis. It is usually the second step in the pipeline, following data understanding, which involves understanding the goals of the analysis, the context in which the data will be used, and the characteristics of the data itself.

Data acquisition involves obtaining data from a variety of sources, such as databases, files, or web APIs, and importing it into a format that can be used for analysis. This may involve cleaning and preprocessing the data to remove errors or inconsistencies, and transforming it into a form that is suitable for the intended analysis.

The specific steps involved in data acquisition will depend on the specific requirements of the analysis and the characteristics of the data. For example, if the data is being acquired from a database, it may be necessary to connect to the database and execute queries to retrieve the desired data. If the data is being acquired from a file, it may be necessary to parse the file and extract the relevant data. Regardless of the specific steps involved, the goal of data acquisition is to obtain and prepare the data for analysis in a way that is efficient and effective.

In general when collecting data there are some base assumption that we need to make and we can't be complete shore about the information that we collect. An example of this can be a sentiment analysis on Twitter that is assumed to represent the entire population, but in reality the analysis come from a sample of those total, only the people that use Twitter, so the statistical relevance is compromise.

2.1.1 Software Tool

Using a programming languages leads to more flexibility, but they are more time consuming to learn and they are complex to understand for some one how is not the author, for the lack of documentation. Using a data management platform is more easy to use and doesn't require a particular study.

- KNIME: general purpose ML tool. It allows performing complex ML tasks via a drug and drop approach
- TALEND: general purpose ETL(extract, transform, load) tool
- AIRFLOW: Python based orchestrator of data science pipeline

It is possible to use tools when all pipeline steps are developed by mans of a tool such as knime or talend, when there are no special constraints in the data acquisition, when there is the need to apply the pipeline multiple times, and when such tools can be used also if you use API. The key point is that there isn't a best solution, is a choice of preference.Using tools like KNIME or Talend can be a good option when developing a pipeline if there are no special constraints on the data acquisition, if the pipeline needs to be applied multiple times, and if the tools can be used with APIs. It's important to note that there isn't a single "best" solution for every situation, and the decision to use a tool like KNIME or Talend will depend on the specific needs and preferences of the user. Ultimately, the key is to choose a solution that works best for the particular problem at hand and meets the needs of the user.

2.1.2 Data acquisition techniques

There are several different methods that can be used to acquire data, depending on where the data is located and the terms of its license. Some common methods include downloading the data directly, accessing the data through an API, or using a scraper to extract the data from a website.

The method that is most appropriate for a given situation will depend on the specific requirements of the analysis and the characteristics of the data. For example, if the data is available for **download** from a website and the license allows it to be used for the intended purpose, then downloading the data directly may be the most straightforward option. If the data is only available through an **API**, then using the API to access the data may be the best approach. If the data is only available on a website and there is no other way to access it, then a **scraper** may be necessary to extract the data.

In some cases, it may not be possible to use any of these methods to acquire the data, due to restrictions on the data or limitations of the tools. In these situations, it may be necessary to use dedicated tools, general-purpose data platforms, or custom Python programs to acquire the data. The specific approach that is used will depend on the specific needs of the analysis and the resources that are available.

API

An API, or Application Programming Interface, is a set of standardized protocols and tools that allows different software systems to communicate with each other. It provides a way for different applications to exchange data and perform specific actions, such as requesting data from a third-party data provider or triggering a particular process in another application. One common type of API is the REST API, which stands for REpresentational State Transfer. REST APIs use the HTTP protocol to transfer information between applications in a standardized, efficient, and secure manner. They are designed to be flexible, scalable, and easy to use, and are widely used to expose data and functionality from web-based systems to external developers and users. APIs can be used for a wide range of purposes, including accessing data from external sources, integrating different applications and services, and building custom applications and integrations. They are an important part of the modern software ecosystem, and are used in many different contexts to enable communication and collaboration between different systems and platforms.

There are four basic operations that can be performed on data: **create**, **retrieve**, **update**, and **delete** (CRUD). These operations allow users to create new data, retrieve existing data, update existing data, and delete data as needed. Most APIs require authentication, which is a process of verifying the identity of a user or application before allowing access to the API. One common way to authenticate API requests is through the use of an API key, which is a unique string of characters that is assigned to a user or application and is used to identify and authorize API requests. API keys are typically provided by the API provider and are required to send requests to the server to retrieve data. API keys may be free to use, or there may be

fees associated with their use, depending on the specific API and the terms of its usage policy. Some APIs may have tiered pricing structures, with different levels of access and functionality available at different price points. Others may offer a limited number of free requests per month, with additional requests available for a fee. It is important to carefully review the terms of use for an API before using it, to ensure that you understand any associated costs and usage limitations.

A REST API exposes a set of public URLs that client applications use to access the resources of a web service. These URLs, in the context of an API, are called **endpoints**.

Data are often in JSON format, so they are organized in document and sub document. (JSON is based on objects and arrays). It work with endpoint and filters in order to manipulate and extract Json documents. To extract content from a Json document there is the need of a query languages. There are no standard query language for Json. KNIME adopts a ≪path expression≫ languages close to XPATH. API are typically static data, but there aren't only static data.

It is important to understand that in many cases, new data is generated continuously, in **real-time**. Remember that real time doesn't meant fast, but means explicit or implicit time constraints. In these cases, using an API to access the data can be problematic, as it may not be able to keep up with the rate at which the data is being produced. This can lead to data loss, as the API may not be able to retrieve all of the data that is being generated. One solution to this problem is to use a collecting tool that sits between the data producer and the consumer, and continuously collects the data as it is generated. This allows the data to be accessed in real-time, even if the producer is generating data faster than the API can retrieve it. A stream processing platform, such as Kafka, can be used to implement this type of collecting tool. Kafka is a distributed streaming platform that is designed to handle large volumes of data in real-time, and can be used to build robust, scalable data pipelines that can handle high-speed data streams. Kafka is written in Java, but it has bindings for many other languages, including Python, which makes it easy to use with a wide range of applications and systems. It is important to note that the use of a collecting tool like Kafka is only possible if the data provider allows it. Some data providers may have restrictions on how the data can be accessed and used, and it is important to carefully review the terms of use before implementing any data collection or analysis systems.

Scraper

When a data provider does not want to share data, or does not provide an API for accessing the data, the only option may be to use a scraper to extract the data from the provider's website. A scraper is a program that automates the process of accessing a website and extracting data from it. Scrapers can be useful for extracting data from websites that do not provide other means of access, or for collecting data from multiple websites in a automated fashion. There are many open-source scrapers available, including browser extensions like "Web Scraper" that can be used to extract data from a single web page. However, these types of scrapers are often limited in their capabilities, and may not work well with more complex websites. In addition, many open-source scrapers are not designed to avoid detection by the website, and may be detected and blocked by the data provider.

To avoid detection, it may be necessary to code a sleep function into the scraper, which causes the scraper to pause for a certain amount of time between requests. This can help to simulate the behavior of a human user, and may make it less likely that the scraper will be detected by the data provider. However, it is important to note that even with these measures, it is still possible for the data provider to detect and block a scraper, and there is no guarantee that a scraper will be able to successfully extract data from any given website.

HTML (HyperText Markup Language) is a markup language that is used to structure and format content on the web. It is the primary language used to define the various elements of a web page, such as headings, paragraphs, lists, and links. HTML uses special tags to describe and structure data, and these tags are used to define the different elements of a web page.

HTML nodes can be used to retrieve web pages by issuing HTTP GET requests and parsing the requested HTML web pages. By default, the output table will contain a column with the parsed HTML converted into XHTML. Then you have to extract data you need by means of regular expression, XPATH or similar. The same solution is available on Knime or in python with the library BeautifulSoup. But also this solution is not enough because in some case you need to go deeper to extract data.

If the page is not in HTML but the server send the code to the page to create the HTML page, in order to scrape we have to simulate the request of the code. The solution in this case is Selenium, it is a library for controlling web browsers through programs and performing browser automation. This is the most difficult case of all.

Chapter 3

Data Storage

3.1 Data Model

Data modeling is the process of designing and creating a model that represents part of the real world in a way that allows data to be stored (write) and queried (read). A data model is a representation of the data structures and relationships that exist within a particular domain, and consists of a conceptual model and a language for describing and querying the data. A data model is typically implemented using a database management system that supports the data model's semantics, or rules for interpreting the data. There are many different types of data models, including relational, hierarchical, network, object-oriented, and document-based models, each of which has its own strengths and weaknesses and is suited to different types of data and applications. Data modeling is an important task in many software development projects, as it allows developers to design and implement a database that is optimized for the needs of the application. Different applications may require different types of interaction with the database, and choosing the right data model is essential for ensuring that the database is able to support the required functionality efficiently and effectively.

The most important features of a data model are:

- Machine readability
- Expressive power
- Simplicity
- Flexibility
- Standardization

3.1.1 The relational model

The relational model is a logical model (independent of the specific implementation). It is a way of organizing data in a database. It is based on the idea of representing data as rows in tables, with each table having a unique name and a set of columns each with a unique name. Tables can be related to each other using foreign keys, which are columns that contain values that refer to the primary key of another table. This allows the data in the tables to be connected and accessed in a logical, structured way.

SQL (Structured Query Language) is a programming language used to interact with databases that follow the relational model. It is used to create, modify, and query the tables and data in a database. SQL allows

users to specify the data they want to retrieve and how they want it organized, and it also provides commands for inserting, updating, and deleting data in the database.

In the relational model 'no information' is an information, and it has different meanings. In all cases where 'no information' is available, the null value is used. It has to be interpreted as a logical value.

Relational model is based on the **closed world assumption**. It refers to the idea that everything that is relevant to a given problem or domain is explicitly represented in the database or system. This means that if something is not present in the database, it is assumed to be false or not applicable.

In the relational model the data is linked by values, in general each table describes one **entity** that has relationships. The relationships could be *one*, *one to meany* ore *many to many*. In case of many to many relationship there is the need to add a new table between entities.

The relational model is implemented in **Relational Database Management Systems** (RDBMS), like mySQL or Oracle. The table model can also be described with other tools, like Excel or CSV.

The relational model has many positive aspects. It is very stricted, due to the close world assumption and the **minimization principle**. The minimization principle states that the amount of data that is stored in a database should be minimized in order to improve the efficiency and performance of the database system. Relational model easily managed by a RDBMS. It also very well know and used and for a large number of task it is still the best option. The RDBMS positive aspects consist in the **ACID** proprieties. The RDBMS positive aspects consist in the ACID proprieties. ACID is a set of properties that are desirable for database transactions in order to ensure data integrity and correctness:

- **Atomic**: This property ensures that a transaction is all or nothing. Either all of the operations within the transaction are completed, or none of them are. This helps to prevent data inconsistencies and corruption.
- **Consistency**: A transaction should leave the database in a consistent state, regardless of whether the transaction is completed or not. This means that the transaction should not violate any of the database's constraints or rules.
- **Isolation**: Transactions should be isolated from each other, so that the effects of one transaction are not visible to other transactions until the first transaction is completed. This helps to prevent conflicts and data corruption.
- **Durability**: Once a transaction has been completed and committed, its effects should be permanent and not subject to loss, even in the event of a system failure or power outage.

Together, these properties help to ensure the integrity and correctness of data within a database. They are particularly important in systems that handle financial transactions, medical records, and other sensitive data.

Advantages of the SQL system could transform in limitations. The fact that the relational model is very stricted make it unusable in some situations. Basically it is not compatible with the modern programming language and it is not able to support cyclical data relationships. In RDBMS is difficult to modify the tables and it is not scalable (it is able to scale up but not scale out). Also in relational models it is very hard to modify existing data.

The speed for execute a query depends on number of row, type of query, algorithm selection and implementation of algorithms and data structure. In general the query that involve more tables are the slowest ones.

3.1.2 NoSQL

NoSQL stands for "Not Only SQL," and it refers to a category of database systems that are designed to handle a wide variety of data models and to scale horizontally across multiple servers. NoSQL databases are often used in situations where the data being stored is too large, too complex, or too diverse to be easily handled by traditional relational database systems. One of the key characteristics of NoSQL databases is that they are **schema-free** or schema-less. This means that the structure of the data being stored in a NoSQL database can vary from one record to the next, and that there is no need to define a fixed schema in advance. This allows NoSQL databases to be very flexible and adaptable, but it also means that they may not offer the same level of data integrity and consistency as relational databases. NoSQL databases also typically operate under the **open world assumption**, which means that they assume that there may be data stored outside of the database that is relevant to the queries being made. This is in contrast to the close world assumption, which is often used in relational databases and assumes that all relevant data is stored in the database. Overall, NoSQL databases are a useful tool for handling large amounts of complex data, but they may not be the best choice for all applications, especially those that require strong data integrity and consistency.

NoSQL models can follow either the **BASE principles** or the **CAP theorem**. The BASE principles are a set of guidelines that describe the behavior of NoSQL databases. They stand for Basic Availability, Soft State, and Eventual Consistency, and they prioritize availability and allow for a certain degree of inconsistency in the short term. The CAP theorem, is a principle that applies to distributed systems, including some NoSQL databases. It states that it is impossible for a distributed system to simultaneously guarantee all three of the following properties: consistency, availability, and partition tolerance. Instead, a distributed system must choose to focus on two of these properties, and possibly sacrifice the third. Some NoSQL databases, such as those that follow the BASE principles, may prioritize availability and partition tolerance, while sacrificing consistency. Others, such as those that follow the CAP theorem, may prioritize consistency and availability, while sacrificing partition tolerance. Overall, the choice of which principles or theorem to follow depends on the specific requirements and constraints of the application.

CAP theorem . The CAP theorem basically it say that

Definition 1. *It is impossible for a distributed computer system to simultaneously provide all three guarantees: Consistency, Availability, Partition tolerance.*

Where:

- **Consistency**: all nodes see the same data at the same time
- **Availability**: every request receives a response about whether it was successful or failed
- **Partition tolerance**: the system continues to operate despite arbitrary message loss or failure of part of the system

a distributed system must choose to focus on two of these properties, and possibly sacrifice the third. In the case of traditional relational database management systems (RDBMS), the focus is generally on consistency and availability (CA). This means that the data stored in the database is always consistent, and the database is always available to accept requests. However, if a partition or failure occurs in the system, the database may not be able to guarantee partition tolerance. On the other hand, many NoSQL systems are designed to focus on either consistency and partition tolerance (CP) or availability and partition tolerance (AP). In a CP system, the data is always consistent, but the system may not be available if a partition occurs. In an AP system, the system is always available, but the data may not be consistent in the event of a partition.

Overall, the choice of which two properties to focus on in a distributed system depends on the specific requirements and constraints of the application.

BASE principles BASE is a set of principles that is often used to describe the behavior of NoSQL databases. The acronym BASE stands for:

- **Basic Availability**: The database is always available to fulfill requests, even if it cannot guarantee strong consistency at all times.
- **Soft State**: The state of the database can change over time, and the database may not enforce strict consistency constraints.
- **Eventual Consistency**: At some point in the future, the data in the database will eventually converge to a consistent state, but there may be a delay before this consistency is achieved. Delayed consistency, as opposed to immediate consistency of ACID properties.
  - Purely a liveness guarantee (reads eventually return the requested value).
  - Does not make safety guarantees.
  - An eventually consistent system can return any value before it converges.

In contrast to the ACID properties of traditional relational databases, which focus on strong consistency and immediate convergence to a consistent state, the BASE principles prioritize availability and allow for a certain degree of inconsistency in the short term. This allows NoSQL databases to scale more easily and to handle large amounts of data, but it also means that they may not offer the same level of data integrity and consistency as relational databases.

NoSQL models

NoSQL databases are a category of database systems that are designed to handle a wide variety of data models and to scale horizontally across multiple servers. There are several different types of NoSQL databases, including:

- **Key-Value Stores**: These databases store data as a set of key-value pairs. The key is used to identify the value, which can be any type of data. Key-value stores are efficient for lookups and are often used to store large amounts of data that does not need to be queried in complex ways. Key value mapping is based hash algorithm very efficient also with respect to data distribution.
- **Column Family Stores**: These databases store data in columns rather than rows, and the key refers to a set of columns rather than a single row. Column family stores are well-suited for storing large amounts of data and are often used for big data applications. Column family store are able to store big data because they distribute data across a large number of servers. The key refers to a set of column.
- **Document Databases**: These databases store data in the form of documents, which are typically formatted in JSON (JavaScript Object Notation). Document databases are designed to be flexible and allow for a wide variety of data structures, and they can be searched using a variety of methods.
- **Graph Databases**: These databases store data in the form of nodes and edges, which can be used to represent entities and the relationships between them. Graph databases are well-suited for storing complex data structures and are often used for applications that involve networking or relationships between data. Nodes and arcs have properties. Nodes can represent entities. The properties are data related to the node or edge. Graphdbs are not able to scale easily.
- RDF databases as well as Tuple stores (not seen in this course)

The choice of which type of NoSQL database to use depends on the specific requirements and constraints of the application.

When you have to decide to describe some model you have to make a choice. When choosing a database model, it is important to consider the specific requirements and constraints of the application, as well as the trade-offs between different models.

Document-based databases, such as those that store data in JSON documents, can be a good choice when you want to access all of the information in a single record at once, since the relationships between data are typically stored within the document itself. This can be more efficient than using a relational database, which may require multiple joins to access related data. Document-based databases are also generally more flexible than relational databases, since they do not require a fixed schema and can accommodate a wide variety of data structures. On the other hand, graph databases, which store data in the form of nodes and edges, are well-suited for applications that involve creating and querying networks or relationships between data. They can be a good choice when the data model is highly interconnected and requires complex queries to navigate the relationships between data. However, graph databases may not be as efficient as other types of databases for storing and querying large amounts of data, and they may not scale as well.

Overall, the choice of database model depends on the specific needs of the application and the types of data and queries that will be performed. It is important to carefully assess the trade-offs between different models and choose the one that is most appropriate for the task at hand. Basically the data model depends on the goal.

3.1.3 Document Database: MongoDB

MongoDB is a popular NoSQL database that is designed to handle large amounts of data and to scale horizontally across multiple servers. MongoDB is a NoSQL database that is known for its flexible schema and ability to store data in a nested, multi-dimensional structure. In MongoDB, each field can contain zero, one, many, or embedded values, and queries can be performed on any field and level. This makes it easy to store and query complex data structures, and it allows the schema to be flexible and easily modified over time. Some key features of MongoDB include:

- Create and manipulate data: MongoDB allows you to create new data by inserting documents into a collection. You can also upload large amounts of data using the mongoimport tool or the bulk write API, and you can update or delete existing data using update and delete methods.
- Query data: MongoDB provides several options for exploring and querying data, including the find method, which allows you to search for documents in a collection, and a variety of query operators, such as $gt and $lt, which allow you to specify conditions for filtering and sorting data.
- Indexing: When working with large datasets, it is often useful to create indexes to improve the performance of queries. MongoDB supports a variety of index types, including single field, compound, and text indexes.
- Nested documents: MongoDB allows you to store nested documents within a parent document, which can be useful for storing complex data structures.
- Aggregation: MongoDB provides the aggregate method, which allows you to perform complex data processing and analysis using pipeline stages. This can be useful for tasks such as group by, distinct, and map-reduce operations.
- Pipelines: The aggregate method in MongoDB uses pipeline stages to allow you to perform complex data processing and analysis. A pipeline is a series of stages that operate on the input data and produce an output. Each stage transforms the input data in some way, and the output of one stage is passed as the input to the next stage.

One advantage of storing data in a nested structure is that it allows for optimal data locality, since related data is stored together within a single document. This can improve performance by reducing the need for

joins and indexes, and it can also make it easier to update or delete related data in a single operation. However, the flexible schema of MongoDB can also present some challenges, such as difficulty in determining the schema or structure of the data, and potential issues with data quality if the schema is not well-defined (or not known) or is changed frequently. When working with MongoDB, it is important to carefully consider the trade-offs between the benefits of a flexible schema and the potential challenges it can present.

3.1.4 Graph Database: Neo4j

A graph model is a way of representing data as a collection of nodes and edges, where nodes represent concepts or entities and edges represent the relationships between them.In a graph first class citizen concepts are modelled as nodes, the way in which nodes are semantically connected is modelled with edges. Graph models are often used to represent complex, interconnected data structures, and they can be applied in a wide variety of fields, including social networks, recommendation systems, and bioinformatics.

In a graph model, the data and schema are typically described in the same way, using nodes and edges to represent the concepts and relationships in the data. This can make it easy to understand the structure and meaning of the data, and it can also make it easier to modify the schema over time as the data evolves. Graph databases are database systems that are designed to store and manage graph data.

In a graph database, there are two main components that are responsible for storing and processing the data: the **storage engine** and the **processing engine**. The storage engine is responsible for managing the underlying data storage, including tasks such as reading and writing data to disk, managing data structures and indexes, and handling transactions and concurrency. The processing engine, on the other hand, is responsible for executing queries and processing data, including tasks such as optimizing queries, executing graph algorithms, and handling data transformations. Together, the storage and processing engines form the core of a graph database and are responsible for the performance, scalability, and reliability of the system. The specific design and implementation of these components can vary depending on the particular graph database and its intended use case.

There are two main types of graph databases: **native** and **non-native**. Native graph databases are optimized for storing and managing graph data, and they typically store attributes and nodes together in a single data structure. Non-native graph databases, on the other hand, store data in a non-graph based model, such as a relational or object-oriented database, but they support a graph query language that allows you to perform graph-based queries on the data.

Overall, graph models and graph databases are powerful tools for representing and managing complex, interconnected data structures, and they are widely used in a variety of applications.

In document based systems a graph can be modelled by means of identifiers that refer to other documents. The main difference between a tree and a graph is that tree doesn't allowed cycles, while graph allowed them.

In a graph model, properties are attributes or characteristics that are attached to nodes or edges in order to add additional information or context to the data. Properties are often represented as key-value pairs, where the key is a unique identifier for the property and the value is the property's data or content. By attaching properties to nodes and edges in a graph model, you can add additional context and meaning to the data, which can make it easier to understand and analyze. For example, you could attach properties to nodes to represent characteristics or attributes of the nodes, such as the name or age of a person in a social network. Similarly, you could attach properties to edges to represent the nature or strength of the relationship between nodes, such as the type of connection or the duration of a friendship in a social network. In some cases, it may be useful to use properties to distinguish between different instances of the same concept or entity. For

example, if you have two nodes in a graph that represent people with the same name, you could use properties such as age or location to differentiate between the two individuals. Similarly, you could use properties on edges to differentiate between different types of relationships, such as strong or weak connections in a social network.

Query languages

One challenge in the field of graph databases is the lack of a standard query language that is supported by all systems. This means that each graph database may have its own proprietary query language, and as a result, it can be difficult to switch between different graph databases or to integrate them with other systems. If you want to use a different graph database or integrate your graph data with other systems, you may need to rewrite your database queries from scratch in the new database's query language. This can be a time-consuming and error-prone process, and it can also make it difficult to migrate or share data between different systems.

Despite the lack of a standard query language, there are several popular query languages that are used by various graph databases. For example, Neo4j, a popular native graph database, uses the Cypher query language, while Gremlin is a popular query language that is supported by several different graph databases.

Overall, the lack of a standard query language in the field of graph databases can be a challenge, particularly if you need to switch between different systems or integrate your data with other systems. However, there are several popular query languages that are widely used by various graph databases, and it is important to familiarize yourself with these languages in order to effectively work with graph data.

Cypher Cypher is a pattern-matching query language that is used to retrieve and manipulate data stored in a graph database, such as Neo4j. One of the key features of Cypher is that it is **expressive** and **declarative**, which means that you describe what you want, rather than how to do it. This makes it easy to write queries that are easy to read and understand, even for people who are not familiar with the details of the database.

In Cypher, you use patterns to specify the nodes and relationships that you want to return in your query. For example, you might use a pattern like `(a:Person)-[:KNOWS]->(b:Person)` to match two people who are connected by a "KNOWS" relationship. The result of a Cypher query is typically a table, which lists all of the nodes and relationships that match the pattern.

It's worth noting that the result of a Cypher query cannot be used to make another query. This is in contrast to other types of databases, such as document databases or SQL databases, where the result of one query can be used as input to another query.

Gremlin Gremlin is a **vertex-based** query language, also known as a graph traversal language. It is designed specifically for querying and manipulating graph data, and is a part of the Apache TinkerPop framework.

Gremlin is a **domain-specific language** (DSL), which means that it is tailored for a specific domain (in this case, querying and manipulating graph data). It is implemented in various graph databases that exhibit certain distribution properties.

In Gremlin, you use expressions to specify a series of traversal steps that you want to take through the graph. These expressions are concatenated together to form a complete query. Gremlin has a number of built-in functions and operators that you can use to filter and transform the data as you traverse the graph.

There are also embeddings of Gremlin in various programming languages, which means that you can use Gremlin to query and manipulate graph data from within your code. This can be useful if you want to build applications that work with graph data and make use of the power of Gremlin.

Polyglot db Some db allow to use different query languages to be interacted, but complex query in a language that is not the one in which the data are stored, it probably will have terrible performance.

3.2 Dataware House

Definition 2. *A data warehouse is simply a single, complete, and consistent store of data obtained from a variety of sources and made available to end users in a way they can understand and use it in a business context.*

A **data warehouse** (DW) is a database designed to support the efficient querying and analysis of data. It is a central repository of data that is used to support business intelligence and decision-making activities. A DW is typically designed to store large amounts of data from various sources, such as transactional databases, log files, and external data sources. The data in a DW is typically organized and structured in a way that makes it easy to analyze and interpret, and it is typically stored in a way that allows for fast querying and retrieval. A DW is often used in conjunction with tools and technologies, such as online analytical processing (OLAP) systems, to enable users to analyze and interpret data in a business context.

A DW is a subject-oriented, integrated, time-varying and non-volatile collection of data that is used primarly in organizational decision making. Subject-oriented refers to the focus of a data warehouse on specific subjects, such as customer data, sales data, or product data. The data in a DW is organized and structured around these subjects, making it easier to analyze and interpret data related to specific business areas.

- Integrated refers to the fact that a data warehouse combines data from multiple sources, such as transactional databases, log files, and external data sources, into a single, consistent view of the data. This allows users to analyze data from different sources in a unified way, rather than having to work with multiple, disconnected data sets.
- Time-varying refers to the fact that a data warehouse stores historical data, as well as current data. This allows users to analyze trends and changes over time, and to make decisions based on this historical context.
- Non-volatile refers to the fact that data in a data warehouse is not updated or deleted as part of normal transactional processing. This makes it possible to store data in a data warehouse for long periods of time without the risk of it being changed or deleted.

The combination of these characteristics makes a data warehouse an ideal tool for organizational decision making, as it allows users to analyze large amounts of data from various sources in a structured, consistent, and historical context.

In general a DW collect data from heterogeneous sources and provide an integrate and shared view of data.

When handling different sources we have two different solution:

- **Data integration** (lazy): query-driven, the integration system is directly connected with clients. Data is combined from multiple sources on-demand, as needed to fulfill a specific query. This means that the integration system is directly connected to the clients, and data is combined and transformed at the time of the query. This can be an efficient solution for simple queries, but can be inefficient for complex queries, as it may require combining and transforming large amounts of data on-the-fly. Additionally,

data integration does not take into account historical data, as it only combines data at the time of the query. This means that it is not possible to analyze trends or changes over time using this approach. Basically it increase the workload.

- **Warehouse** (edger): stores data that has been pre-integrated and pre-processed from multiple sources. The data are integrated before they are used. This means that the data is already structured and organized in a way that makes it easy to analyze and interpret. A data warehouse is designed to support complex, history-based queries, and allows users to analyze trends and changes over time. Additionally, a data warehouse is not directly connected to the clients, which means that it can store large amounts of data without being impacted by the workload of querying and transforming the data. Overall, a data warehouse is a more efficient and effective solution for complex, history-based queries and analysis, compared to data integration.

.

Having a data warehouse has many advantages, but also some cons to consider. In particular in the creation phase the warehousing process could be very long and involves multiple areas of a company. Usually the integration of a DW allow discovering a huge data quality issues, also assess and improve data is not an easy task. The new data sources to integrate or new type of queries can impose long and costly redesigns of DW.

Despite the aspects just seen, DW implementation is mainly positive for a company. In particular for read data it allow fast complex query, and there is no interference with operational sources. In addition data can be modified and redesigned, sore historical series is now possible and it allow a better access control. *Data warehouse is still used as first data analytic tool in a large number of companies.*

3.2.1 Data warehouse architecture

Data warehouse architecture refers to the overall design and structure of a data warehouse, including the hardware, software, and processes that are used to collect, store, and access data. There are several different types of data warehouse architectures, each with its own unique features and benefits. Some common types of data warehouse architectures include the following:

- **Single-tier architecture**: In a single-tier architecture, the data warehouse and all of its components, such as the database, ETL tools, and reporting tools, are all housed on a single server. This is the most basic type of data warehouse architecture and is suitable for small organizations with simple data needs.
- **Two-tier architecture**: In a two-tier architecture, the data warehouse is separated into two tiers: a front-end tier and a back-end tier. The front-end tier contains the user interface and reporting tools, while the back-end tier contains the database and ETL tools.
  This architecture is more scalable than single-tier architecture and is suitable for medium-sized organizations. The advantages of this architecture are the fact that it allows users to have high quality data even if is not possible to access original data sources, it also allow that complex queries does not impact the data sources. At a logical level the DW is based on a multidimensional model, while all data sources are defined with relational model. DW also implements specific techniques for supporting large amount of data.
- Three-tier architecture: In a three-tier architecture, the data warehouse is separated into three tiers: a front-end tier, a middle tier, and a back-end tier. The front-end tier contains the user interface and reporting tools, the middle tier contains the ETL tools and other data processing components, and the back-end tier contains the database. This architecture is more scalable and flexible than two-tier architecture and is suitable for large organizations with complex data needs.

- **Hub-and-spoken**: In a hub-and-spoke architecture, the data warehouse is organized around a central hub, with satellite data marts connected to the hub. The hub contains the master copy of the data, and the satellite data marts contain copies of the data that have been tailored to meet the specific needs of different departments or business units. This architecture allows for decentralized data management, as each department or business unit can manage its own data mart, while still having access to the central hub for a complete view of the data.
- **Centralized DW**: A centralized data warehouse is a type of data warehouse architecture in which all of the data is stored and managed in a central location. This architecture is suitable for organizations with a small number of users and a relatively simple data environment. A centralized data warehouse can be implemented using a single-tier, two-tier, or three-tier architecture, depending on the needs of the organization. It is more a logical approach than a real one

The type of data warehouse architecture chosen will depend on the size and complexity of the organization, as well as its data needs and resources.

Reconciled data are operational data that have been created after an integration and quality process of data sources. They are a main advantage of the two-tiered architecture, as they provide a clear separation between the extraction phase and the feed phase. Reconciled data are integrated, coherent, correct, up-to-date, and detailed, and can be used as a reliable source of information for business operations and decision making. Reconciled data are the main advantage in the two tiers architecture, with the clear separation between extraction phase and feed phase.

Data Mart A data mart is a subset or aggregation of data in the primary data warehouse (DW). It includes a set of data that is relevant for a specific business area or type of business. Data marts derived from the primary DW are referred to as **dependent** data marts. In large organizations, this solution is adopted to increment the development strategy. The primary DW defines the boundaries of relevant information for specific business needs and related queries, and the data mart has reduced dimensions with respect to the DW, which increases efficiency. In some cases, data marts are directly fed by data sources and are referred to as **independent** data marts. While the lack of a primary DW can make the design process easier, the design of the schema itself can be complex, and conflicts may arise in the case of multiple independent data marts.

In some cases, it is possible to have independent data marts, where each organizational unit develops its own data mart independently. However, this architecture can lead to a lack of coherence in data due to differences in measurement and dimension. Independent data marts are often used in organizations where each business unit or department has its own data needs and wants to have control over its own data mart. This approach can be useful for smaller organizations or for departments that have unique data requirements that cannot be met by the primary DW.

One solution to this problem is the use of a **data mart bus**, which integrates data marts in a logical way using common dimensions. Another solution is **federation**, which involves both physical and logical integration between data marts. This solution is ideal in high dynamic contexts, such as mergers and acquisitions, but can be critical during the integration phase.

3.2.2 Multidimensional model

Once data has been cleansed, integrated, and transformed, there is the need to figure out how to get the most information out of it. There are essentially three different approaches supported by as many categories of tools, to querying a DW by end users:

- **Reporting**: it is a tool-based approach to querying a data warehouse that allows users to generate pre-defined, structured reports. These reports are typically simple and straightforward, and do not require IT knowledge to use. Reporting tools are user-oriented and are designed to provide clear, repeated reports that are easy to understand. The structure of the reports is always the same, making it easy for users to understand and interpret the data.
- **OLAP**: Online analytical processing (OLAP) is a tool-based approach to querying a data warehouse that allows users to analyze and explore the data interactively. OLAP tools are designed to support the multidimensional analysis of data, and typically require the user to think in a multidimensional way and to have a basic understanding of the interface of the tool being used. OLAP is the main way that users interact with the information contained in a data warehouse, and it allows users to build complex analysis sessions in which each step taken is a consequence of the results obtained in the previous step. OLAP is organized in active work sessions, and is suitable for users who need to perform in-depth analysis of the data. Very complex queries are not an issue. OLAP is orientated toward non-IT users.
- **Data mining**: is a tool-based approach to querying a data warehouse that allows users to discover patterns and relationships in the data. Data mining tools require the user to have a basic understanding of the principles that underlie the tool being used, and typically involve the use of algorithms and statistical techniques to analyze the data. Data mining is suitable for users who need to perform detailed analysis of the data, and can be used to identify trends, patterns, and relationships that might not be immediately apparent through other means.

An OLAP session consists of navigation path that reflects the analysis process of one or more facts of interest under different aspects and at different levels of detail. This path takes the form of a sequence of questions often formulated not directly, but by difference with respect to the previous question. Each step of the analysis session is marked by the application of an OLAP operator which transforms the last query formulated into a new query. The result of the queries is multidimensional; OLAP tools typically represent data in a tabular fashion highlighting different dimensions using multiple headings, colors, etc. OLAP-type navigation is based on a multidimensional data model.

The **multidimensional model** is the foundation for representing and querying data in data warehouses. The facts of interest are represented in **cubes** where:

- each cell contains numerical measures which quantify the fact from different points of view
- each axis represents a dimension of interest for the analysis
- each dimension can be the root of hierarchy of attributes used to aggregate the data stored in the base cubes

OLAP has many operators that allowed to interact with the cubes. *Slice* is an operation that fix a value for one specific dimension from a given data cube and provides a new sub-cube (it reduce the dimensionality) *dice* is an operation that selects two or more dimensions from a given data cube and provides a new sub-cube (it reduce the set of data). When we have data in the form different from the needed, then the *aggregation* methods can be applied to the attributes to obtain the desired attributes. The *Roll-Up* increase the data aggregation by removing a level of detail form a hierarchy. The *Drill-Down* decrease data aggregation by introducing an additional level of detail into a hierarchy. Finally, *Pivoting* change the way of representation with the aim of analysing the same information form different points of view.

Drill-Across creates a join between two cubes.

3.2.3 Dimensional Fact Model

The **dimensional fact model** (DFM) is a graphical model that is used to design data marts, which are smaller, more focused data warehouses that are designed to support specific business processes. The DFM is designed to:

- effectively support the conceptual design
- create an environment on which to intuitively formulate user quires
- allow dialogue between the designer and the end user to refine the specifications of requirements
- create a stable platform from which to start the logic project (regardless of the target logic model)
- return expressive and unambiguous a posteriori documentation

The conceptual representation generated by the DFM consists of a set of **fact schemas**, which model the basic elements of a data mart, including facts, measures, dimensions, and hierarchies.

DFM basic constructs A **fact** is a concept of interest for decision making, typically models a set of events that occur in the enterprise. It is essential that a fact has dynamic aspects, that is, it evolves over time. A **measure** is a numerical property of a fact and describes a quantitative aspect of interest for the analysis (for example, each sale is measured by its receipts). A **dimension** is a finite-domain property of a fact and describes an analysis coordinate (typical dimensions for sales fact are product, store , date). A fact describes a n-to-n relationship among dimensions.

The general term **dimensional attributes** means the dimensions and any other attribute, always with discrete values, which describe them (for example, a product is described by its type, by the category to which it belongs, by its brand, by the department in which it is sold).

A **hierarchy** is a directed tree whose nodes are dimensional attributes and whose edges model may-to-one associations between pairs of dimensional attributes. It contains a dimension, located at the root of the tree, and all the dimensional attributes that describe it.

To sum up facts are the numerical data that are stored in the data mart, and measures are the values that are calculated from the facts. While dimensions are the categories or attributes by which the facts are organized, and hierarchies are the relationships between the different levels of a dimension.

DFM advanced constructs A **descriptive attribute** contains additional information about a dimensional attribute of hierarchy, to which it is connected by a one-to-one association. it is not used for aggregation because it has continuous values and/or because it comes from a one-to-one association. Same arcs of the fact schema be be optional.

A **cross-dimensional** attribute is an attribute, dimensional or descriptive, whose value is determined by the combination of two or more dimensional attributes, possibly belonging to distinct hierarchies. Two dimensional attributes can be connected by two or more distinct directed paths, provided that each of them still represents a functional dependency (**convergence**).

Shared hierarchy is an abbreviation used to denote that a portion of the hierarchy is replicated multiple times in the schema.

A **multiple arc model** is a many-to-many association between two dimensional attributes. An **incomplete hierarchy** is a hierarchy in which, for some instances, one or more levels of aggregation are absent (because they are not known or not defined).

3.2.4 Star/snowflake schema

To convert a dimensional fact model (DFM) into a **logical model** involves a number of steps which start with the conceptual scheme and produce a logical schema for the data mart as the final output. The inputs to this process include the conceptual scheme, the workload (the types of queries and other operations that will be performed on the data mart), the data volume (the amount of data that will be stored in the data mart), and the system constraints (such as hardware and software limitations).

During the logic design phase, you can apply a set of principles that are different from those used in operational systems in order to optimize the schema for the specific workload and constraints of the data mart. This often involves introducing data redundancy and denormalizing relationships in order to improve query performance. The resulting logical schema can then be used as the basis for the physical design and implementation of the data mart.

The main operation to be performed during logical design are:

- choice of logical schema to use (star/snowflake schema)
- translation of conceptual schemes
- choice of dynamic hierarchies
- choice of views to materialize
- applying other forms of optimization

It can result in to a star schema or in to a snowflake schema:

- **Star schema**: it consists of a single fact table surrounded by a number of dimension tables. The fact table contains the numerical data that is being analyzed, and the dimension tables contain the categories or attributes by which the data is organized.
- **Snowflake schema**: it is a variant of the star schema that further normalizes the dimension tables. In a snowflake schema, the dimension tables are further broken down into sub-tables, which can be linked back to the main dimension table. This allows for more granular data storage and more efficient querying, at the cost of increased complexity and slower performance when compared to a star schema.

The basic rule for translating a fact schema into a star schema is to: *create a fact table containing all the measures and descriptive attributes directly related to the fact and, for each hierarchy, create a dimension table containing all its attributes.* In addition to this simple rule, the correct translation of a fact schema requires a thorough discussion of advanced DFM constructs.

There are conflicting opinions on the usefulness of snowflaking because it generate contrast with the data warehousing philosophy, it represents an unnecessary 'beautification' of the scheme and it is inadequate or otherwise more complex to support. Despite that it may be of some use when the ratio between the cardinalities of the primary and secondary dimension table is high, as it determines a strong saving of space. However, the disk space saving must be evaluated taking into account the performance impact due the possible additional cost (and complication) of the joins. The snowflake could be useful when a proportion of a hierarchy is common to more dimensions (the secondary dimension table is used for more hierarchies).

3.2.5 Dynamic dimensions

Hierarchies, which represent the relationships between the different levels of a dimension, are properly modeled by functional dependencies in a dimensional fact model (DFM) or a data warehouse schema. Functional dependencies specify the relationships between attributes in a database, such that the value of one attribute determines the value of another attribute. For example, in a hierarchy representing the organizational struc-

ture of a company, the value of the attribute "department" might determine the value of the attribute "employee."

However, functional dependencies imply a static view of reality, which can be problematic when modeling hierarchies because they can change over time. For example, an employee might be promoted to a new position in a different department, which would change the functional dependency between the "department" and "employee" attributes.

To accommodate the dynamics of hierarchies in the logical design of a data warehouse, it is important to consider how the dynamic nature of the hierarchy will be used by queries. In some cases, it may be necessary to store historical values of the hierarchy attributes in order to support querying across time. This can be done by only considering events (such as sales transactions or other business processes) that refer to unchanged values in the hierarchy, and by storing the value of the hierarchy attributes at the instant in time when the event occurred.

Chapter 4

Data Preparation

4.1 Data integration and enrichment

Data integration is the process of combining data from multiple sources into a single, unified view. This can involve extracting data from different sources, transforming it into a common format, and loading it into a central repository such as a data warehouse. The goal of data integration is to make it easier to analyze and understand the data by bringing together all of the relevant information in one place.

Data enrichment is the process of adding additional context or information to data. This can involve adding external data sources to provide more context, or using algorithms and machine learning techniques to derive insights and conclusions from the data. The goal of data enrichment is to make the data more useful and valuable by adding additional layers of meaning and understanding.

A **homogeneous model** is a model that is constructed using data from a single source or domain. The data used to build the model is assumed to be similar in structure and format, and the model is designed to operate on this specific type of data. On the other hand, a **heterogeneous model** is a model that is constructed using data from multiple sources or domains. The data used to build the model may have different structures and formats, and the model is designed to operate on a wide range of data types. Heterogeneous models can be more robust and flexible than homogeneous models, as they are able to handle a wider range of data inputs. However, they may be more complex and require more resources to build and maintain. Both homogeneous and heterogeneous models have their own advantages and trade-offs, and the best approach will depend on the specific needs and goals of the application.

In the context of data integration, **heterogeneity** refers to the differences in structure, format, and content of data from different sources. This can include differences in the types of data (e.g. numeric, categorical, text), the names of the data attributes (e.g. "customer_id", "ID", "id_num"), and the definitions of the data attributes (e.g. "customer_id" might refer to a unique identifier in one dataset, and to a customer's age in another). Conflict and correspondence are terms that are often used in the context of data integration to describe the relationships between data from different sources. Conflict refers to cases where data from different sources contradicts or contradicts each other, while correspondence refers to cases where data from different sources is consistent or matches up. Heterogeneity, conflict and correspondence are synonyms in the following.

- **Name heterogeneity**: it refers to differences in the naming conventions used for data attributes across different sources. For example, one dataset might use the attribute "customer_id" to refer to a

unique identifier, while another dataset might use the attribute "ID" for the same purpose. It can be divided in:
  - *Synonyms*: different names for the same concepts (employee, clerk)
  - *Homonyms*: same name for different concepts ("City" as city of birth in one schema, as city of residence in another schema)
  - *Hyperonimies*: two concepts related by an IS-A relationship
- **Type heterogeneity**: it refers to differences in the types of data used by different sources. For example, one dataset might use numeric data to represent customer ages, while another dataset might use text data (e.g. "under 18", "18-24", etc.) to represent the same information. The same concept is represented with different conceptual structures in two schema.
  - Different definition domains for the same attribute in two schema
  - Attribute in one schema and derived value in another schema
  - Attribute in one schema and entity in another schema
  - Attribute in one schema and generalization hierarchy in another schema
  - Entity in one schema and relationship in another schema
  - Different abstraction levels for the same concept in two schema: e.g. two entities with homonym names related by an IS-A hierarchy in two schema
  - Different granularities in the definition domains
  - Different cardinalities in the same relationships
  - Key conflicts
- **Model heterogeneity**: it refers to differences in the models or frameworks used by different sources to represent and interpret data. For example, one dataset might use a linear regression model to predict customer purchasing behavior, while another dataset might use a decision tree model for the same purpose.

4.1.1 Phases of the methodology, inputs, outputs, and methods used

Lets now explore the methodology of schema integration.

1. Schema transformation (or Pre-integration)
   Input: n source schemas
   Output: n source schemas homogenized
   Methods used: model transforming and reverse engineering
2. Correspondences investigation
   Input: n source schemas
   Output: n source schemas and correspondences
   Methods used: techniques to discover correspondences
3. Schemas integration and mapping generation
   Input: n source schemas and correspondences
   Output: integrated schema and mapping rules between the integrated schema and input source schemas
   Methods used: new classification of conflicts and conflict resolution transformations

Pre integration

Pre-integration strategies refer to techniques that are used to prepare data for integration, before it is combined with other data sources. There are two main types of pre-integration strategies: binary and N-ary.

- **Binary** pre-integration strategies involve combining two data sources into a single integrated dataset. There are two main types of binary pre-integration strategies:
  - *Composed*: This strategy involves combining two data sources by concatenating or appending one dataset to the other. This is useful when the data sources have similar structures and formats, and can be easily merged together.
  - *Balanced*: This strategy involves balancing the data from two data sources by selecting a subset of each dataset that is representative of the entire dataset. This is useful when the data sources have different sizes or structures, and a balanced dataset is needed for further analysis or integration.
- **N-ary** pre-integration strategies involve combining three or more data sources into a single integrated dataset. There are two main types of N-ary pre-integration strategies:
  - *One-step*: This strategy involves combining all of the data sources into a single dataset in a single step. This is useful when the data sources have similar structures and formats, and can be easily merged together.
  - *Iterative*: This strategy involves combining the data sources in an iterative process, gradually adding more data sources to the integrated dataset as needed. This is useful when the data sources have different structures and formats, and require more complex integration processes.

Correspondences investigation

Correspondence investigation is a process that is used to determine how data from different sources can be integrated and combined. One important aspect of correspondence investigation is *semantic relativism*, which refers to the idea that different data sources may have different schemas, or structures, for organizing and representing data. This can make it challenging to combine data from different sources, as the data may have different attribute names, data types, and definitions.

To address this problem, it is often necessary to create a schema of correspondences, which is a map or guide that shows how data from different sources can be matched up and integrated. The schema of correspondences can include information about how data attributes from different sources are related, and can be used to transform and standardize the data as needed to facilitate integration.

The process of creating a **schema of correspondences** typically involves comparing the schemas of the different data sources and identifying any discrepancies or mismatches. This can involve manually reviewing the data to identify correspondences, or using automated tools or algorithms to assist with the process. Once the schema of correspondences has been created, it can be used to transform and standardize the data as needed, and to integrate the data into a single, unified dataset.

4.1.2 Schemas integration and mapping generation

When integrating data from multiple sources, it is common to encounter conflicts or discrepancies between the data. These conflicts can take many forms, such as differences in attribute names, data types, definitions, or meanings. To address these conflicts, it is often necessary to use transformation or integration rules that specify how the data should be transformed or reconciled. A *rich language for expressing correspondences* is a tool or framework that allows users to specify and describe the relationships between data attributes from different sources. This can include information about how data attributes are related, and can be used to create transformation or integration rules that specify how the data should be transformed or reconciled.

We first have to provide: the **new classification of types of conflicts**, and **corresponding conflict resolution transformation** (also called integration rules).

New classification of conflicts

There are many different types of conflicts that can arise when integrating data from multiple sources. A new classification of these conflicts can be helpful in organizing and understanding these conflicts, and in developing strategies for resolving them.

- *Classification conflicts* refer to discrepancies or mismatches in the classification or grouping of data attributes. For example, one data source might classify customers into age groups (e.g. "under 18", "18-24", etc.), while another data source might classify customers into income groups (e.g. "low income", "middle income", etc.).
- *Descriptive conflicts* refer to discrepancies or mismatches in the descriptions or definitions of data attributes. For example, one data source might define a "customer_id" attribute as a unique identifier for each customer, while another data source might define the same attribute as a customer's age.
- *Structural conflicts* refer to discrepancies or mismatches in the structures or formats of data attributes. For example, one data source might store customer addresses as a single string, while another data source might store the same information as separate attributes for each address component (e.g. street, city, state, zip code).
- *Fragmentation conflicts* refer to discrepancies or mismatches in the way that data is partitioned or divided into separate datasets. For example, one data source might store customer data in a single dataset, while another data source might store the same data in multiple datasets, each representing a different customer group or time period.
- *Instance level conflicts* refer to discrepancies or mismatches in the values or instances of data attributes. For example, one data source might store a customer's age as 25, while another data source might store the same customer's age as 26.

These are just a few examples of the types of conflicts that can arise when integrating data from multiple sources. There may be many other types of conflicts, depending on the specific data sources and the contexts in which the data is used.

Mapping rules

Mapping rules are used to specify how data from different sources should be transformed or reconciled when integrating data. These rules are defined between the integrated schema, which is the final structure or format of the combined data, and the source schemas, which are the structures or formats of the individual data sources. To create mapping rules, it is necessary to examine the correspondences and conflict resolution transformations that are involved in the integration process for each source schema. This may involve reviewing the data from each source, comparing the structures and formats of the data, and identifying any discrepancies or mismatches. Once the correspondences and conflicts have been identified, mapping rules can be defined to specify how the data from each source should be transformed or reconciled to fit the structure of the integrated schema. These rules may include instructions for renaming attributes, converting data types, combining or separating data attributes, and other types of transformations. The mapping rules for each source schema can then be used to transform the data from that source into the desired format, and to integrate it with the data from other sources. This process may involve using automated tools or algorithms, or may require manual review and interpretation by a human expert. The resulting integrated dataset will have a consistent structure and format, and will be ready for further analysis or use.

Basically mapping rules are defined between the integrated schemas and the source schemas. This rules can be found for the source schema S looking at correspondences and conflict resolution transformations involved adopted for S in the integration process.

Instance Level Conflicts Given two sources S_1 and S_2, let A_{1k} and A_{2k} represent the same attributes of a real world object represented as t_1 in S_1 and as t_2 in S_2:

- *Attribute conflicts*: An attribute conflicts occurs if $t_1.A_{1k} \neq t_2.A_{2k}$
- *Key conflicts* (entity or tuple conflicts): A_{1k} is primary key for t_1 and A_{2k} is primary key for t_2. A Key conflict occurs if $t_1.A_{1k} \neq t_2.A_{2k}$ and $t_1.A_{1i} \neq t_2.A_{2i}$ or all $i \neq k$

Techniques that deal with instance-level conflicts can be applied in two different phases of the life cycle of a data integration system, namely, at *design time* and at *query time*. In in both cases, the actual conflicts occur at query time; however, the design time approaches decide the strategy to follow for fixing conflicts before queries are processed, i.e., at the design stage of the data integration system. The techniques operating at query time incorporate the specification of the strategy to follow within query formulation.

Resolution function A **Resolution function** takes two (or more) conflicting values of an attribute as input and outputs a value that must be returned as the result to the posed query. Common resolution functions are MIN and MAX. For non numerical attributes, further resolution functions can be identified, such as CONCAT.

4.2 Data Integration more in depth

Data integration refers to the process of combining data from different sources into a single, cohesive view. There are two primary forms of data integration:

- **Deduplication**: This refers to the integration of data within a single table. It involves identifying and eliminating duplicate records within the table, in order to ensure that there is only a single, accurate record for each entity.
- **Integration of data in multiple tables**: This refers to the process of combining data from multiple tables into a single, cohesive view. It involves creating relationships between the tables in order to allow data from one table to be related to data in another table. This is often done using a process called normalization, which involves breaking down data into smaller, more manageable chunks in order to reduce redundancy and improve data integrity.

Both forms of data integration are important for ensuring that data is accurate, consistent, and easily accessible. They are often used in conjunction with each other in order to create a comprehensive and accurate view of an organization's data.

4.2.1 Deduplication

In deduplication generally records refer to the same real world object if they differ only in one letter in Name & Surname, the same if records differ only in one letter in Name & Surname (although in different languages).

In order to handle deduplication we have to proceed with an assessment. **Data quality assessment** is the process of evaluating the quality of data by checking for various characteristics such as accuracy, completeness, consistency, and validity. The goal of data quality assessment is to identify and correct any problems or errors in the data that may affect its usefulness and reliability. **Data fusion**, on the other hand, is the process of combining multiple data sources into a single, more comprehensive dataset. The goal of data fusion is to improve the quality and accuracy of the data by taking advantage of the strengths of each

individual data source. There is a strong correlation between data quality assessment and data fusion, as the quality of the data being fused will significantly impact the quality of the resulting dataset. Therefore, it is important to perform a data quality assessment on each individual data source before attempting to fuse them, in order to ensure that the resulting dataset is as accurate and reliable as possible.

4.2.2 Data integration of two tables

Some key rules in dis case are that records are the same if they differ only in one letter in Name & Surname or the name in one table is a known contraction of the name in the second one and the place of birth is the same.

A **data quality improvement** in the data integration of two tables refers to any changes or enhancements made to the data in the tables that result in an improvement in the overall quality of the data. This can include a variety of different activities, such as correcting errors or inconsistencies in the data, standardizing the data to a common format, or adding missing or incomplete data. There are many different techniques that can be used to improve the quality of data during the integration process, and the specific techniques used will depend on the nature of the data and the specific goals of the integration project. Some common techniques for improving data quality during integration include:

- Data cleansing: This involves identifying and correcting errors and inconsistencies in the data, such as incorrect or inconsistent values, duplicates, or missing data.
- Data standardization: This involves converting the data to a common format, such as standardizing dates or numbers to a specific format, or converting data to a common case (e.g., upper or lower case).
- Data enrichment: This involves adding additional data or context to the data, such as adding geographic information or demographic data.
- Data deduplication: This involves identifying and removing duplicate data records, in order to ensure that each record represents a unique entity.

Overall, the goal of data quality improvement in data integration is to ensure that the resulting integrated dataset is as accurate, consistent, and complete as possible, and that it can be used effectively for the intended purposes.

Record Linkage

Record linkage is the process of identifying records in a dataset that refer to the same real-world entity, despite potentially having different identifiers or information. This can be a challenging task, as the records may have different spellings, formatting, or other discrepancies that make it difficult to determine if they refer to the same entity. Record linkage is often used to merge or deduplicate datasets, or to link records across different datasets in order to gain a more complete and accurate view of an entity. For example, record linkage might be used to combine customer records from different departments or business units in order to create a single, comprehensive customer profile.

There are many different techniques and algorithms that can be used to perform record linkage, including probabilistic methods, deterministic methods, and machine learning approaches. The specific technique used will depend on the nature of the data and the specific goals of the record linkage process.

Lets see some existing techniques:

- **Empirical**: are based on statistical analysis of the data itself, and do not rely on any prior knowledge about the data or the entities being linked. These techniques are typically based on the assumption

that records that refer to the same entity will have similar or identical data values, and they use various statistical measures to determine the likelihood that two records refer to the same entity.

- **Probabilistic**: use statistical models to estimate the probability that two records refer to the same entity, based on the similarity of the data values in the records. These techniques typically involve estimating the probability of a match based on the probability of certain data values occurring together, and using this information to calculate the overall probability of a match. (Assume I know a sample of frequencies of "distances" between pairs of tuples, I may decide those matching and non matching ones in the universe by projecting such knowledge on the sample on the universe of pairs).
- **Knowledge based**: decision based on rules that have to be matched by pairs of tuples. Rely on domain-specific knowledge or rules to determine if two records refer to the same entity. These techniques typically involve applying a set of rules or heuristics to the data in the records, and using the results of these rules to determine the likelihood of a match.
- **Mixed** probabilistic and knowledge based: combine elements of both probabilistic and knowledge-based techniques, using statistical models and domain-specific knowledge to determine the likelihood of a match. These techniques can often provide more accurate results than either probabilistic or knowledge-based techniques alone, as they can take advantage of the strengths of both approaches.

Probabilistic record linkage Lets now see in detail the steps that compose the probabilistic record linkage.

1. *Preprocessing*: Normalization of formats
2. *Blocking*: Blocking with search space reduction
3. *Compare*: Choice of distance function
4. *Compare*: Find a sample of pairs of tuples that are known to be matching (i.e. represent the same observable) or not matching (i.e. represent different observable). Evaluate for each value of distance the frequency of matching and non matching
5. *Decide*: Evaluate the distance for pairs in the universe. Analyzing the distribution of step 4, choose the threshold d_{min} and d_{max}
6. *Decide*: For pairs of tuples in the universe
   - whose distance d is $d < d_{min} \rightarrow$ decide for matching
   - whose distance d is $d > d_{max} \rightarrow$ decide for non matching
   - whose distance d is $d_{min} < d < d_{max} \rightarrow$ decide for possible match

Normalization means that record are reorganized according to a common format. In order to chose a **distance function** we have to make two choice: select the set of attributes involved in the function, and select the function for distance.

4.2.3 Data fusion conflict handling strategies

There are several strategies to asses data fusion conflict.

- **Conflict ignoring**: this strategies do not make a decision as to what to do with conflicting data and sometimes are not even aware of data conflicts.
  - *Pass It On*: escalates conflicts to user or application
  - *Consider All Possibilities*: creates all possible value combinations
- **Conflict avoiding**: this strategies acknowledge the existence of possible conflicts in general, but do not detect and resolve single existing conflicts. Instead, they handle conflicting data by applying a unique decision equally to all data.
  - *Take The Information*: prefers values over null values

  - *No Gossiping*: returns only consistent tuples
  - *Trust Your Friends*: takes the value of a preferred source
- **Conflict resolution**: this strategies do regard all the data and metadata before deciding on how to resolve a conflict. They can further be subdivided into:
  - **Deciding**: when they choose a value from all the already present value. Like *Keep Up To Date*: takes the most recent value
  - **Mediating**:when they choose a value that does not necessarily exist among the conflicting values. Like *Meet In The Middle*: takes and average value

Chapter 5

Advanced Data Management

5.1 Data and Information quality dimensions from Data to Big Data

5.1.1 Data quality

The quality is fitness for use. The quality of data can affect decisions made on them. Data quality is not only on rows but it also refers to columns (attributes), we have to have the right attributes of the domain in order to analyse it correctly without bias. The mantra is *"garbage in, garbage out"*. Each year companies spend 600 billions for bad, inaccurate, dirty or missing data.

The source of data is important, we have to trust the source in order to trust the data. Some times the data quality is on the consumer, not the producer. The data quality tradeoff is a matter of usefulness versus the faithfulness.

- **Usefulness**: precision of internal representation on image with reference to the task performed.
- **Faithfulness**: how faithfully the ground truth about the data is displayed.

Lets now see some fundamental concepts which we will discuss more in detail later. **Quality** are the characteristics of an artifact that bear on its ability to satisfy stated or implied user needs and expectation. Characteristic or **dimension** is a specific characteristic describing the quality of information, usually not measurable. Sometimes classified in terms of sub-dimension. In order to define**metrics** we refer to the definitions given within the ISO standard 9126-1 and the ISM3 framework: a set of elements encompassing both a *measurement procedure* and a proper *unit of measure*. Where a measurement procedure (or method) is an algorithm that takes the element to measure and associates it with a measure (be it ordinal or interval value), and a unit of measure (or scale) is the domain of values returned by the measuring procedure. In general, several metrics can be associated to each quality dimension.

5.1.2 Data quality dimensions

It is even difficult to understand how many dimensions data quality has. For example data can be:

- **Inaccurate**: If data is inaccurate, it means that it is not correct or reliable. This can occur for a variety of reasons, such as mistakes in data entry, errors in measurement or calculation, or problems with the data collection process.
- **Incomplete**: If data is incomplete, it means that it is not fully representative or lacking key information. This can occur when data is not collected or recorded properly, or when certain pieces of information are missing or left out. Incomplete data can lead to incomplete or biased conclusions, and can make it difficult to accurately understand or analyze a situation.
- **Inconsistent**: If data is inconsistent, it means that it is not in agreement or does not match up with other data. This can occur when data is collected or recorded in different ways, or when there are errors or inconsistencies in the data collection process.

There are a huge quantity of measures and dimensions but we will focus on the following ones:

- **Accuracy, Correctness, Precision**: Proximity of data in representing a given reference.
  - *Accuracy*: refers to how closely the data represents the true value or reality.
  - *Correctness*: refers to whether the data is free from errors or mistakes.
  - *Precision*: refers to how consistently the data is collected or measured, and how many digits are used to express it.
- **Completeness, Pertinence**: Data represent all (and only) the aspects of the reality of interest.
  - *Completeness*: refers to whether all relevant data has been collected and recorded.
  - *Pertinence*: refers to whether the data is relevant and appropriate to the task at hand.
- **Minimality, Redundancy, Compactness**: Data represent all the aspects of the reality of interest.only once and with the minimal use of resources.
  - *Minimality*: refers to the use of the minimum amount of data necessary to accurately represent a subject or solve a problem.
  - *Redundancy*: refers to the presence of duplicate or unnecessary data.
  - *Compactness*: refers to the use of the least amount of space or storage to represent the data.
- **Consistency, Coherence**: Data comply to all the properties of their membership set (class, category,...) as well as to those of the sets of elements the reality of interest is in some relationship.
  - *Consistency*: refers to whether the data is recorded or collected in the same way each time.
  - *Coherence*: refers to whether the data is logical and makes sense within the context in which it is used.
- **Readability, Comprehensibility, Usability**: Data are easily perceivable and understood by users.
  - *Readability*: refers to how easy it is to read and understand the data.
  - *Comprehensibility*: refers to how easy it is to understand the meaning of the data.
  - *Usability*: refers to how easy it is to use the data to solve a problem or make a decision.
- **Accessibility, Usability**: Data can be effectively "exploited" (consumed) by users.
  - *Accessibility*: refers to how easy it is to obtain or retrieve the data.
  - *Usability*: refers to how easy it is to use the data to solve a problem or make a decision.
- **Currency, Volatility, Timeliness**: Data are up-to-date.
  - *Currency*: refers to how up-to-date or current the data is.
  - *Volatility*: refers to how quickly the data changes or becomes outdated.
  - *Timeliness*: refers to whether the data is available when it is needed.

Another issues is that the measures can be both objective or subjective.

- **Objective measures**: formal and precise ways to measure metrics for a quality dimension in terms of the values of a domain. Independent of human perception/assessment.

- **Subjective measures**: ways to measure metrics for a quality dimension depending on the perception (usually explicit and ordinal rating) of people involved in the measurement process. Dependent on human perception/assessment.

5.1.3 Data quality dimension in the relational model

The data quality dimensions and metrics are:

- Accuracy
- Completeness
- Currency and Timeliness
- Consistency
- Tradeoff between dimensions

Accuracy

Accuracy con be defined on alphanumeric values, on tuples, and on relations, but not on numeric values.

Definition 3. *Accuracy of a value v is defined as the closeness between v and a value v' considered as the correct representation of the real world phenomenon that v aims to represent.*

Since usually we do not know, and it could be very costly to know, "the correct representation of the real world phenomenon that v aims to represent", two kinds of accuracy can be identified, namely syntactic accuracy and semantic accuracy.

- *Syntactic accuracy*: it is defined when the vocabulary or reference domain of values is known (e.g. the domain of Italian first names, the domain of Czech municipalities), and is the degree to which values correctly represent the domain values of underlying vocabularies or reference domains. It is finding a way to the closest string that can be referring to the wrong one. An example can be the soundness and it is about the sound of the pronounce of a string in order to find the simile ones, hypnotizing that the misspelling doesn't change the sound to much.
- *Semantic accuracy*: is defined as the degree to which data values correctly represent the real world facts.

Metrics for syntactic accuracy There are two types of metrics: one based on alphanumeric strings (`<<Santa Margherita Ligure>>`), and one based on their structure in terms of items/token (`<<Santa>> <<Margherita>> <<Ligure>>`). We will now focus on **Metrics based on strings** which adopt a distance function, measured between the data value and the values of a reference domain. In particular we will see Edit Distance metric.

- **Un-normalized Edit Distance**: UED between $v1$ and $v2$ is equal to the number of *insertions* plus the number of *deletions* plus the number of *substitution* of alphanumeric symbols necessary to transform $v1$ into $v2$.
$$D(v1, v2) = \sum_{i=1}^{|v1|} [v1_i \neq v2_i]$$
- **Normalized edit distance**: ED_{norm} between $v1$ and $v2$ are values in the domain D, where the maximum number of symbols is n, is minimum number of character to move from a string to another.
$$ED_{norm}(v1, v2) = 1 - ED(v1, v2)/n$$

Values are between 0 and 1.

The normalized Edit Distance does not measure the distance but its complement to 1, i.e. the accuracy of $v1$ with respect of $v2$, considered a certified value: Accuracy = 1 corresponds to Distance = 0. It work well with short strings, but not with longs one (es. movie names). In general with accuracy we have to try to make the interval between 0 and 1.

Completeness

Completeness con be defined on alphanumeric values, on numerical values, on tuples, on attributes, on relations and on objects. Completeness of a dataset is the coverage with which the observed phenomenon is represented in the dataset. It is the presences of null values.

- Tuple completeness: presence of null values in a tuple
- Attribute completeness: presence of null values in an attribute
- Table completeness: number of non null values in a table

This definition assumed a *closed world assumption*: all that is represented in the BD is true, all the rest is false. This is the typical assumption that is made in DBMSs. An alternative hypothesis is that of an *open world*: nothing is known about everything that is not represented. In this case we introduce the **object completeness**, which takes into account the fact that the objects that can be represented are more than the tuples of the table, and an indirect estimate is needed for this carnality.

- Object completeness: number of non null values in all the objects represented in tuples

Temporal quality dimension

We will no show two temporal quality dimension : currency and timeliness.

Currency The currency measures how quickly the data is updated (compared to the corresponding phenomenon int the real world) A first measure of the currency is the time delay between the time $t1$ of the event in the real woeld that caused the variation of the data, and the instant $t2$ of its registration in the information system. The measure is expensive because generally the event is not known.

Currency can be also seen as the difference between the time of arrival at the organization and the time in which the update is performed. Measurable if there is a log of arrivals and updates.

Can be also seen as difference form the "last update" metadata. For values with known update frequency, currency calculable in approximate but inexpensive way.

Timeliness Timeliness measures how up-to-date the data is with respect to a particular process (or processes) using it. Timeliness, unlike currency, depends on the process, and is associated with the time point in which it must be available for the process that uses the data. There may be data with high currency, but now obsolete for the process that uses them. If the timetable was produced outside the maximum time and was loaded immediately after being produced, it is current but not timely.

Consistency

There are two meanings of consistency:

1. Consistency of the data with the integrity constraints defined on the schema. (ex. ZIP Code must be consistent with City)
2. Consistency of the different representations of the same object of reality present in the database. (ex. the address must be represented with the same format in all the databases in which it is defined.

Basically any violation of an integrity constraint defined on the database. Consistency constraints are restrictions that are placed on data in a database to ensure that it is consistent and coherent within the system. For example, a consistency constraint might be placed on a database table to ensure that all values in a particular column are unique.

Business rules, on the other hand, are rules established by an organization that govern how data is managed and used. For example, a business rule might specify that all customer names must be written in uppercase, or that invoices must be issued within 30 days of the purchase date.

In summary, consistency constraints are restrictions that are placed on data in a database to ensure its consistency, while business rules are rules established by an organization to govern how data is used.

Quality tradeoff

Consistency and completeness in the relational model can be irreconcilable when one wants to respect referential integrity. In the statistical/economic/financial domain (e.g. increase in GDP in the last three months) there is a tradeoff between timeliness and accuracy/competence, favoring the latter. On the web, in order to "arrive first" with news, in many cases timeliness is preferred over accuracy and completeness (e.g. weather forecast, agency dispatch).

5.1.4 Quality assessment and improvement

The goal of the improvement phase is to clean data in such a way to increase their quality with reference to values assessed in the previous phase. Since quality is multidimensional, specific activities are needed for the different qualities. In the improvement phase we can adopt two general strategies, namely **data driven** and **process-driven**. Data-driven strategies improve the quality of data by directly modifying the value of data through the comparison with other data considered of good quality. For example, obsolete data values are updated by refreshing a database characterized by higher currency.

5.2 Advanced data management

5.2.1 Big data

Big Data is not just data, there are a few new consideration. Firstly we have to separate Big Data from Open Data.

- **Big Data**
  - *Volume*: Data at rest. Terabytes to exabytes of existing data to process.
  - *Velocity*: Data in motion. Streaming data, milliseconds to seconds to respond.
  - *Variety*: Data in many forms. Structured unstructured, text, multimedia.
  - *Veracity*: Data in doubt. Uncertainty due to data inconsistency and incompleteness, ambiguity, model approximations.

- **Open Data**
  - *Visibility*: Data in the open. Open data is generally open to anyone. Which raises issues of privacy, security and provenance.
  - *Value*: Data of many values. Large range of data values from free (data philanthropy to high value monetization).

Analyze big data on a single "big" server is slow, costly and hard to realize, basically is not feasible. The solution is distributing the analysis over a set of inexpensive hardware commodities, called parallel system. As we now scale up is limited, while scale out is theoretically unlimited. This parallel system can rise some main issues like:

- *Synchronization*: is the process of coordinating the actions of multiple processes or threads in order to access shared resources in a parallel system. Without synchronization, it is possible for two or more processes to simultaneously access the same resource and cause conflicts or errors.
- *Deadlock*: is a situation that can occur in a parallel system where two or more processes are blocked and unable to proceed because they are waiting for a resource that is held by another process. This can create a cycle where each process is waiting for a resource held by another process, resulting in a standstill.
- *Bandwidth*: refers to the amount of data that can be transmitted over a communication channel in a given amount of time. In a parallel system, the bandwidth of the communication channel can be a limiting factor in the performance of the system, as the processes may have to wait for the channel to become available before they can transmit data.
- *Coordination*: refers to the process of coordinating the actions of multiple processes or threads in a parallel system. This can include synchronization, as well as other techniques such as load balancing and resource allocation.
- *Failure*: in a parallel system can occur when one or more of the processes or threads experiences an error or stops functioning properly. This can have a cascading effect on the rest of the system, depending on the nature of the failure and the design of the system.

One of big data architecture basic features is the linear scalability (scale out). In this case computation task moves towards data and not vice-versa. It work on commodity hardware and it need a failure management.

5.2.2 HDFS

HDFS, or Hadoop Distributed File System, is a distributed file system designed to store large amounts of data across a network of commodity hardware. It is used to support the storage and processing of data for applications running on the Apache Hadoop platform, which is a framework for distributed processing of large data sets.

HDFS is designed to be fault-tolerant, scalable, and flexible, and it is used in many big data applications where large amounts of data need to be processed and analyzed. It allows data to be stored on multiple nodes in a cluster, and it provides a way to access this data through a network interface. HDFS is optimized for large files, and it has features such as data replication and automatic failure recovery to ensure the reliability and availability of data stored in the system.

The Hadoop Distributed File System (HDFS) has a master-slave architecture, where one or more servers act as a master node, and a set of worker nodes store and process the data. The master node in HDFS is called the *NameNode*, and it is responsible for managing the filesystem namespace and regulating access to files by clients. The NameNode maintains a record of all the files stored in HDFS, as well as the blocks that make up each file and the locations of those blocks on the worker nodes. The worker nodes in HDFS

are called *DataNodes*, and they are responsible for storing the actual data blocks and replicating them as needed. Each DataNode communicates with the NameNode to report on the blocks it stores and to receive instructions about changes to the filesystem. In addition to the NameNode and DataNodes, HDFS also includes a *Secondary NameNode*, which is a helper node that performs periodic checkpoints of the filesystem metadata and helps to prevent data loss in the event of a NameNode failure. HDFS is designed to be highly fault-tolerant, and it uses replication to ensure that data is always available, even if one or more nodes fail. The replication factor, which is configurable, determines how many copies of each data block are maintained in the system. This helps to ensure that the data is always accessible, even if one or more nodes fail or become unavailable.

It allowed many file format like: text/csv, JSON, SequenceFile, binary key/value pair format, Avro, Parquet, ORC, optimized row columnar format.

5.2.3 Apache Spark

MapReduce

MapReduce is a programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster. Programmer defines two functions, *map* and *reduce*.

$$Map(k1, v1) \rightarrow list(k2, v2)$$

That takes a series of key/value pairs, processes each, generates zero or more output key/value pairs.

$$Reduce(k2, list(v2)) \rightarrow list(v3)$$

That executed once for each unique key $k2$ in the sorted order; iterate through the values associated with that key and produce zero or more outputs.

System "shuffles" data between map and reduce (so "reduce" function has set of data for its key) automatically handles system failures, etc.

MapReduce is a powerful programming model and implementation for processing large data sets in a parallel and distributed way. However, it is not always the best solution for every problem. One issue with MapReduce is that not all problems can be easily described or implemented using the MapReduce model. MapReduce is well-suited for problems that can be expressed as a series of independent, parallelizable tasks, but it may not be as effective for problems that require a more complex or iterative approach. Another issue with MapReduce is that it relies on persistence to disk, which can be slower than in-memory processing. MapReduce writes intermediate results to disk after each map and reduce step, which can add significant overhead and slow down the processing. In some cases, an in-memory approach may be more efficient, especially if the data fits in memory and the processing can be done entirely in memory. Despite these limitations, MapReduce is still widely used for many big data processing tasks, and it has been a key component of the Apache Hadoop ecosystem. It is a valuable tool for handling large data sets in a distributed way, and it has proven to be effective for many types of problems.

Apache Spark

An alternative of MapReduce is Apache Spark, a general-purpose processing engine. The processing engine, instead of just *map* and *reduce*, defines a large set of operations (trasformations and actions). Operations

can be arbitrarily combined in any order. Apache Sparck is an open source software that support Java, Scala and Python. The key construct is *Resilient Distributed Dataset* (RDD).

Apache Spark supports data analysis, machine learning, graphs, streaming data, etc. It can read/write from a range of data types and allows development in multiple languages.

The software component of Spark Apache consists of several key components:

- Spark Core: This is the underlying execution engine for Spark, and it is responsible for scheduling, distributing, and monitoring tasks on the cluster. Spark Core includes APIs for programming entire Spark applications in Java, Scala, Python, and R.
- Spark SQL: This component provides a SQL-like interface for working with structured data in Spark. It includes a dataframe API for working with tabular data, as well as support for reading and writing data from a variety of sources (such as databases, filesystems, and object stores).
- Spark Streaming: This component allows Spark to process data in real-time, by breaking it into small batches and processing each batch as it arrives. Spark Streaming supports input from a variety of sources, including Apache Kafka, Flume, and TCP sockets.
- Spark MLlib: This is a library of machine learning algorithms and utilities that can be used with Spark. It includes support for classification, regression, clustering, collaborative filtering, and other common machine learning tasks.
- GraphX: This is a library for graph processing and analysis that can be used with Spark. It includes support for creating and manipulating graphs, as well as algorithms for graph analysis and computation.

In all cases, Spark runs as a library within your program, and it provides APIs in multiple programming languages (Java, Scala, Python, and R) for interacting with the system. One of the key features of Spark is its ability to run tasks either locally (on a single machine) or on a cluster of machines. This allows it to scale to very large data sets and to handle a wide range of workloads. Spark can be deployed in a number of different ways, including standalone mode (where it manages its own cluster), as well as on top of cluster managers such as Apache Mesos or YARN. In order to access data stored in external storage systems, such as HDFS or Amazon S3, Spark uses the Hadoop InputFormat API. This allows it to read data stored in a variety of formats, such as Avro, Parquet, and others, and to process it using the parallel processing capabilities of Spark. Overall, Spark is a flexible and powerful platform for data processing and analytics, and it is widely used in a variety of applications.

Apache Spark can be run in a standalone mode, where it manages its own cluster and resources. This is useful for testing and development, as well as for small-scale data processing tasks. To run Spark in standalone mode, you can pass "local" or "local[k]" as the master URL when you create a SparkContext in your application. It debug using local debuggers. Great for development and unit tests.

DataFrames

A DataFrame is a distributed data structure in Apache Spark that is organized into named columns. It is similar to a table in a relational database, or a data frame in Python's pandas library or R's DataTables. DataFrames are immutable, which means that once they are created, their contents cannot be modified. However, they do have a lineage, which means that you can trace the transformations that were applied to create a DataFrame and use that information to optimize distributed computations.

There are several ways to construct a DataFrame in Apache Spark. One way is to read data from external files, such as CSV or Parquet files, using the Spark SQL API. You can also create a DataFrame by transforming an existing DataFrame (either in Spark or pandas), by parallelizing a Python collection (such as a list), or by applying transformations and actions to an existing DataFrame.

5.2.4 Handoop

Apache Hadoop is an open-source software framework for storing and processing large amounts of data in a distributed and parallel manner. It was developed based on research at Google on the MapReduce programming model and the Google File System (GFS).

Hadoop consists of two main components: the **Hadoop Distributed File System** (HDFS) and the **MapReduce** programming model. HDFS is a distributed file system that is designed to store large amounts of data across a network of commodity hardware, and it is optimized for large files and high-throughput access. MapReduce is a programming model and implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

The Hadoop architecture consists of a cluster of machines, with each machine running a Hadoop daemon. There are two main daemons in Hadoop: the *NameNode*, which manages the filesystem namespace and regulates access to files by clients, and the *DataNode*, which stores the actual data blocks and replicates them as needed. In addition to the NameNode and DataNode, Hadoop also includes a *Secondary NameNode*, which is a helper node that performs periodic checkpoints of the filesystem metadata and helps to prevent data loss in the event of a NameNode failure.

Clients submit MapReduce jobs to the Hadoop cluster, and the job is divided into tasks that are executed by the worker nodes in the cluster. The intermediate results are stored on the worker nodes and are brought together by the reducer tasks to produce the final output. Hadoop is designed to be fault-tolerant, and it can recover from failures by reexecuting tasks on other nodes if necessary.

Overall, Hadoop is a powerful and widely used platform for storing and processing large amounts of data in a distributed and parallel manner. It is used in many big data applications and has become a key component of the modern data landscape.

Apache Hadoop 2.0 is a software framework for storing and processing large amounts of data in a distributed and parallel manner. It introduced several significant changes and improvements over the previous version, including the addition of support for *YARN* (Yet Another Resource Negotiator), which is a resource management platform that allows multiple applications to share a common cluster. One of the applications that can be run on Hadoop 2.0 with YARN is Apache *Accumulo*, which is a distributed key-value store that is optimized for high-performance data processing and analytics. Accumulo is based on Google's BigTable design, and it provides features such as cell-level access control, versioning, and support for real-time reads and writes. Another application that can be run on Hadoop 2.0 with YARN is Apache *HBase*, which is a distributed, column-oriented database that is built on top of HDFS. HBase provides low-latency access to large amounts of data, and it is often used for real-time analytics and online applications. Apache Spark is another application that can be run on Hadoop 2.0 with YARN. *Spark* is a fast and flexible data processing and analytics platform that is designed for in-memory processing and supports a wide range of applications, including batch processing, stream processing, machine learning, and SQL. Overall, Hadoop 2.0 provides a powerful and flexible platform for running a variety of data processing and analytics applications, and it is widely used in many big data and analytics scenarios.

5.2.5 Data Lake

The term "data lake" was coined 2010 by James Dixon. He distinguished between the approach to manage data on *Handoop* and *data marts/warehouses*. A **data mart** is a smaller subset of an organization's data that is specifically designed for fast querying and analysis. It typically contains a subset of the organization's data that is relevant to a particular business unit or group, and it is designed to support the specific reporting

and analytics needs of that group. A **data lake** is a large, centralized repository that allows businesses to store all their structured and unstructured data at any scale. The data stored in a data lake can come from a variety of sources, including transactional databases, social media feeds, log files, and sensor data. The data is typically stored in its raw, unprocessed form, and it is only transformed and prepared for analysis when needed. One key difference between a data mart and a data lake is the scope of the data they contain. A data mart typically contains a smaller, more focused subset of the organization's data, while a data lake is a central repository for all of the organization's data. Data marts are typically used for specific business functions or groups, while data lakes are used for a wider range of analytics and data processing tasks. Another difference is the level of structure in the data: data marts typically contain more structured, organized data, while data lakes contain a mix of structured and unstructured.

An **enterprise data warehouse** (EDW) is a central repository of an organization's structured data, designed for fast querying and analysis. It is typically used to support large-scale, enterprise-wide data processing and analysis, and it is built using specialized software and hardware to support high performance and scalability. One of the key challenges facing EDWs today is the increasing volume, variety, and velocity of new data sources. With the proliferation of digital devices and the rise of the Internet of Things (IoT), organizations are generating and collecting large amounts of data from a wide range of sources. This data is often unstructured, and it may arrive at high speeds, making it difficult to manage and process using traditional EDW technologies. Another challenge is the cost of licenses and maintenance for EDW software and hardware. These costs can be significant, and they may limit an organization's ability to scale or upgrade its EDW. Finally, there is a growing recognition that the traditional EDW paradigm, which emphasizes structured, centralized data storage and batch processing, may not be well-suited to the needs of modern data-driven organizations. As a result, there is a shift towards more flexible and scalable data architectures, such as data lakes and cloud-based data platforms, which can support a wider range of workloads and data types. Overall, the enterprise data warehouse landscape is changing rapidly, and organizations need to be mindful of these trends and challenges as they plan and manage their EDW investments.

In a **data lake**, the approach to storing and organizing data is often referred to as a "bottom-up" approach. This means that data is ingested into the data lake in its raw, unstructured form, and it is only transformed and prepared for analysis when needed. This allows organizations to store a wide range of data types and sources in the data lake, without the need to define the schema of the data upfront. By contrast, in an enterprise data warehouse (EDW), the approach to storing and organizing data is often referred to as a "top-down" approach. This means that the schema of the data is typically defined upfront, and the data is transformed and cleaned to fit this schema as it is loaded into the EDW. The schema is typically designed to support the specific reporting and analytics needs of the organization, and the data is typically organized in a way that allows for efficient querying and analysis. One key difference between the two approaches is the level of flexibility and agility they offer. The bottom-up approach of the data lake allows organizations to store and process data more quickly and easily, without the need to define the schema upfront. This can be particularly useful when dealing with large amounts of data, or when the data sources or types are not well-known in advance. However, it can also make it more difficult to understand and work with the data, as the schema is not defined until the data is queried or analyzed. The top-down approach of the EDW, on the other hand, provides more structure and organization to the data, which can make it easier to work with and understand. However, it also requires more upfront planning and schema design, which can be time-consuming and may limit the flexibility of the EDW. Overall, the choice of approach will depend on the specific needs and goals of the organization, and it is important to carefully consider the trade-offs between flexibility and structure when designing a data architecture.

A data lake architecture typically consists of the following components:

- Data Ingestion: This component is responsible for collecting data from various sources, such as transactional databases, log files, and sensor data, and storing it in the data lake.
- Data Storage: This component is responsible for storing the raw data collected from various sources in the data lake. Data can be stored in a variety of formats, such as structured, semi-structured, and unstructured.
- Data Processing: This component is responsible for transforming the raw data stored in the data lake into a format that can be queried and analyzed. This may involve using tools such as Apache Spark or Apache Flink to process the data.
- Data Catalog: This component is a metadata repository that stores information about the data stored in the data lake, such as data definitions, data lineage, and data quality.
- Data Access: This component is responsible for providing users with access to the data stored in the data lake. This may involve using tools such as SQL or business intelligence (BI) tools to query the data.
- Data Governance: This component is responsible for enforcing policies and procedures around the use and management of data in the data lake. This may include defining roles and permissions for different users, as well as implementing security measures to protect the data.

It's worth noting that the specific components included in a data lake architecture can vary depending on the specific needs and requirements of an organization.