DATA LIFECYCLE

Describe data lifecycle according to CRISP-DM methodology

Cross-industry Standard Process, better known as CRISP-DM, is an open standard process model that describes the common approaches used by data mining experts.
Data lifecycle is composed of **six phases not necessarily sequential**. In most of the cases, in fact, movement back and forth between phases is needed.
The six phases are:

1. **Business understanding**: a project plan is developed using the information gathered from the analysis process. Objectives are understood and the user's (or client's) problem is translated into a data mining problem.
2. **Data understanding**: this phase implies a more focused examination of the available data for data mining, in order to avoid unexpected issues during the next step. Data understanding implies the **access to data** and the following **analysis** using tables and charts to asset data quality and describe the results in the project documentation.
3. **Data preparation**: this phase comprehends all the necessary activities to **create the final dataset** (attribute/feature selection, eventual variable transformation and data cleaning). Depending on the objectives, data preparation implies: merging data records, choosing a sample subset, record aggregation, creation of new attributes, sorting data for modeling, handling missing or null values, splitting the dataset into train and test data.
4. **Data modeling**: modeling is typically performed through **multiple iterations**. Several models are usually tested with default parameters before manually adjusting them or returning to the data preparation phase for modifications required by the selected model.
5. **Evaluation**: after completing data modeling phase (which implies that the models are technically correct and effective), the models are analyzed to verify their **precision** and **robustness** in order to meet user's expectations.
6. **Deployment**: this is the final phase and the developed model must be made **available to users**. A **report** is written, or a data mining system is implemented for direct user interaction. Generally, the deployment phase includes two key activities: results planning/monitoring and completing summary operations such as drafting a **final report** and **verifying the project**.

Data management definition

Data Management refers to the processes and practices used to ensure that data is properly collected, stored, and maintained throughout its lifecycle.

DATA ACQUISITION

Definition of data acquisition

Data acquisition is the process of **collecting and importing data** for analysis. The steps involved depend on the specific analysis requirements and the characteristics of the data. When acquiring data, certain assumptions must be made, such as the fact that we can never be entirely certain about the accuracy and completeness of the information we collect.

Data acquisition techniques

There are several methods for acquiring data, which depend on **where data is located** and the **licensing terms** associated with. Those methods are: direct download, API and web scraping.

- **Direct download** is the easiest method and should always be the first option when possible. The issue with it is that the owner/provider of data often does not share it easily.
- **API** (Application Programming Interface) is a **set of standardized interfaces** which allow different software systems to **communicate** with each other. APIs enable applications to **exchange data** and perform specific actions, providing authorized users access to the data through system interactions that generate outputs.
- **Web scraping** involves using algorithms to **simulate user activity** on a web page and extract all the visible information while attempting to avoid detection and blocking (legality of web scraping is still unclear).

API techniques

API (Application Programming Interface) is a set of standardized interfaces which allow different software systems to communicate with each other. APIs enable users to **request data from third-party** providers, granting access to relevant information. APIs typically handle static data.
A common API type is **REST API** (REpresentational State Transfer), which uses an **HTTP protocol** to transfer information between applications in a standardized, efficient, and secure manner.
They are developed in order to be **flexible, scalable** and **easy to use**. Key commands are **CRUD operations**:

- **Create**
- **Retrieve**
- **Update**
- **Delete**

Most APIs require authentication and a common method to achieve that is to use an **API key**, a unique string of characters assigned to a user to **authorize API requests**. APIs can be free, freemium (offering limited access for free with additional data available for purchase), or fully paid.
REST APIs provide a set of URLs, called **endpoints**, which clients use to access a web service's data. The result of an API request (which includes filters and endpoints) is often returned as a JSON file, a format based on objects and arrays. There is no universal query language for JSON, but tools like KNIME can be used for processing. One limitation of APIs is that they may face **challenges in handling real-time data**.

Why does Kafka guarantee speed?

Kafka guarantees speed because APIs, when handling real-time data (which does NOT necessarily mean fast but involves explicit and implicit temporal constructs), can encounter some problems that lead to data loss. Kafka can be useful for this type of data collection.

Kafka is a distributed platform for data streaming and it's designed to **handle large volumes of real-time data**. With Kafka, it is possible to **modify or transform data during its flow**, **eliminating the need to rely on an external service for storage and processing**. This is precisely why Kafka ensures high-speed data collection. However, Kafka can only be used if the data provider allows it.

Web scraping techniques

If a data provider does not want to share its data, the only solution is to extract it from web pages using a scraper. There are various web scraping techniques, such as **browser extensions** (e.g., *Web Scraper*), but these are limited in capability, struggle with complex web pages, and do not evade website detection effectively. To avoid detection, **sleep functions** between requests can be useful. More advanced scrapers use **HTML**, a markup language that structures web pages using tags to define elements. To extract data efficiently, **regular expressions** or **XPATH** can be used. This can be implemented in KNIME or in Python using the *BeautifulSoup* library. In some cases, an additional step is required, especially when dealing with **dynamic web pages** that load content asynchronously. In such scenarios, *Selenium*, a library that allows automation and control of web browsers through programming, is the only viable solution.

DATA STORAGE

Description of data modeling

Data modeling is the process of **design and development** of a model that represents part of the real world in a way that can be **stored (write)** and **queried (read)**.
A data model is a **logical representation of data structures** and existing **relationships** of a particular domain. It consists of a conceptual model and a language to describe and query data.
A data model is typically implemented using a DBMS that supports the model's **semantics**. The most famous and popular data models are:

- **relational**, based on the entity-relationship model;
- **NoSQL**, which tries to overcome the limitations of the relational model. NoSQL models are:
    - **key-value stores**
    - **column family stores**
    - **document-based**
    - **graph**

Each of these ones has its own characteristics and features and the choosing of the model depends on the research's goals.
Most important features of a data model are: **machine readability, expressive power, flexibility, simplicity** and **standardization**.

Relational model

A relational model is a logical framework for structuring data in a database. It's based on the concept of **representing data as rows** (records) within **tables**, where each table has its unique name and **columns** (**attributes**) with distinct names.
Tables can be linked using **foreign keys**, which are columns with values that reference the primary key in other tables. Typically, each table describes an entity that can have relationships classified as *one-one*, *one-many* or *many-many*.
In the relational model, **"no information" is still information**. It follows the ***closed world assumption***, which assumes that everything considered relevant is contained in the database. Otherwise, it's false or not relevant.
Another key principle is the **minimization principle**, which dictates that stored data should be minimized in order to ensure efficiency.
The standard language to interact with relational databases is **SQL** (Structured Query Language).
Relational databases are effectively managed by **RDBMS**, which offers advantages such as a **strict schema** and the adherence to **ACID properties**:

- **Atomicity**: ensures that a transaction is ***all or nothing***: either the entire operation is successfully completed or nothing is executed. This prevents data inconsistency and corruption.
- **Consistency**: a transaction must leave the database in a consistent state, regardless if it is completed or not. This means that **transactions must not violate any database rule** or constraint.
- **Isolation**: transactions must be isolated from each other. In this way, **the effects of one transaction are not visible to others** until it's fully completed. This prevents conflicts and corruption.
- **Durability**: once a transaction is fully completed, the **effects must be permanent** and not subject to loss, even in the event of a system crash.

These properties ensure the **correctness** and **consistency** of a database, treating each modification as a single unit of operations, which leads to reliable and predictable results. The ACID properties are a fundamental feature of RDBMS.
However, the same characteristics that make relational databases robust can also present limitations:

- An attribute can only hold a single value (**no multi-value attributes**).
- They **lack compatibility** with modern object-oriented programming languages.
- They **do not support cyclic relationships**.
- Once implemented, the schema is **difficult to modify**.
- They support scale-up (vertical scaling) but **not scale-out** (horizontal scaling), making them less suitable for distributed architectures.

NoSQL model

NoSQL (Not Only SQL) refers to a huge variety of models designed to **scale horizontally across multiple servers**. They are typically used in situations too complex to be well handled by a relational database. Being **schema-free** is one of the main characteristics of NoSQL models. Schema-free means that the **schema (not strict) can change** according to the specific record or data.
These models are based on the ***open world assumption***, that means even data outside the database can be relevant for the problem (in the closed world assumption of the relational model, if something is not stored in the database, it's assumed to be false).
NoSQL models allow **great flexibility** and **complexity**, but they might not be the best solution in case integrity and consistency are strongly required.
NoSQL models follow either **BASE principles** (Basic Availability, Soft State, Eventual Consistency) and the **CAP theorem** (a distributed system can ensure only two of three CAP components: Consistency, Availability, Partition Tolerance).

Pros and cons of NoSQL schema-free
PROS:

- **Adaptability**
- **Flexibility**
- **Horizontal scalability: high performances and availability**

CONS:

- Less integrity and consistency
- Querying data becomes slow and complex
- Can't normalize data, which can make storage more complex

CAP Theorem

CAP Theorem is a principle applied to **distributed systems** (including some NoSQL databases).
The theorem states that it is impossible, for a distributed system of computers, to ensure simultaneously:

- **Consistency**: all the nodes see the data at the same time.
- **Availability**: all the requests must receive a response (positive or negative).
- **Partition tolerance**: the system continues to operate even in the case of the loss or failure of a part of it.

In general, RDBMSs follow the CA approach, while NoSQL models prefer either CP or AP.

BASE principles

For certain situations, a model based on the ACID properties (Atomicity, Consistency, Isolation, Durability) might be too strict. So, some NoSQL databases are based on a **less strict model**, following the BASE principles:

- **Basic Availability**: the database is **always available for the requests** even if it is not possible to guarantee strong consistency all the time. This is possible by using a **distributed approach** for database management.
- **Soft State**: the state of the database can **change over time** (being schema-free, there are **different schemas for different data**). The database can't impose strict consistency constraints.
- **Eventual Consistency**: at some point in the future, data in the database will converge to a state of consistency, but there is no guarantee as to when. So, when new data is added to the system, they will gradually **propagate one node at a time**, until the entire system becomes consistent. This is in contrast to the *immediate consistency* of ACID properties.

Types of NoSQL models

- **Key-value stores**: data are stored as a set of key-value pairs. **Key** is used to **identify the value**, which can be every type of data. These kinds of databases are really efficient to retrieve data and are often used to store **large amounts of data that don't need complex queries**. They are based on **hash algorithms** and are efficient in data distribution.
- **Column family stores**: these databases store data in columns rather than rows, where the key refers to a set (family) of columns. They are often used in Big Data fields because they are **distributed across multiple servers.**
- **Document-based**: this kind of database stores data in the form of documents, which are typically in JSON or BSON format. They are **flexible** and can handle a **huge variety of data structures.**
- **Graph**: they store data as **nodes and edges**, which can be used to represent **entities** and **relationships** between nodes. They fit for complex structures and are often used for networking, yet they are **not very scalable.**

Describe the document-based model

Document-based is a NoSQL database model known for its **flexible schema** and the possibility of storing data in **multidimensional nested structures**, called **documents.**
Its schema is flexible and **editable over time** because there is **no need for a strict, fixed structure.** Documents can be linked using **references.** This makes the document-based model particularly useful for representing **hierarchies**.
An example of a document-based database is **MongoDB**, where data is stored in BSON format. Data in MongoDB is accessible using **indices** and each document contains zero, one or more **embedded values**. Queries can be done for each level of the document.
In MongoDB, every document must be stored within a **collection**, which in turn is stored within a database. Data can be created and manipulated simply by inserting documents into collections. The stored data can be freely manipulated, even in large volumes, and can also be loaded from external sources.
The most important feature of MongoDB is its ability to perform **aggregation queries**. To execute these, the aggregate operation is used, introducing the concept of a **pipeline**.

A pipeline is a sequence of aggregation operations applied to data in stages. Each stage processes documents from the collection and passes the results to the next operator until the final result is produced. Additionally, optional parameters can be provided to refine the aggregation process.

Describe graph database and the differences with the document-based (with examples)

A graph model represents data as **nodes and edges**, where **nodes** represent **concepts or entities**, and **edges** represent **relationships** between them. Additionally, **properties** (attributes attached to nodes or edges) can store extra information, helping specify the strength of a relationship or differentiate instances of the same concept or entity.
Graph models are commonly used for representing complex and **interconnected structures**, with applications in social networks, recommendation systems, and bioinformatics. The data structure is embedded within the data itself, making it intuitive and adaptable, as the schema can evolve over time with the data.
A graph database consists of two main components: the **storage engine**, which manages data storage, and the **processing engine**, which executes queries and processes data. Together, these components determine performance, scalability, and readability.
There are two types of graph databases:

- **native graph databases**, optimized for storing and mapping data in a graph format;
- **non-native graph databases**, which store data in a non-graph model but support a graph-based query language.

One major challenge with graph databases is the **lack of a standard query language**, making it difficult to switch between different graph databases and sometimes requiring starting from scratch. Some common graph query languages include Cypher (used in Neo4j), which is a declarative, pattern-matching language, meaning users specify what they want rather than how to retrieve it. Query results are typically tables, but they cannot be used as inputs for other queries. It allows users to define a series of traversal steps, which are then concatenated to form complex queries. Some databases, known as polyglot databases, support multiple query languages, but complex queries tend to perform poorly.
The **key difference** between a graph model and a document model lies in their data structures. In document databases, values contain structured or semi-structured data, known as documents, while in graph models, data is structured as nodes and relationships, where nodes store properties and can be labeled with one or more tags, and relationships are named and directional.
A fundamental distinction between graph and document models is the **handling of cycles**. Document models use tree-like structures, which do not allow cycles, while **graph models naturally support cycles**, making them ideal for complex, highly connected data. Unlike document databases (which easily distribute data and handle large volumes efficiently), **graph databases struggle with scalability.** Graph models do not scale as easily, making them less suited for massive distributed architectures compared to document-based systems like MongoDB.

Data warehouse

A data warehouse is a **single, complete and consistent repository** of data obtained by various sources and made available to end users so they can understand and use it for business purposes.
A DWH is a database specifically developed to **efficiently support queries and analysis** and is typically organized and structured to facilitate operations. It is often used alongside tools and technologies such as OLAP (Online Analytical Processing).
DWH is a collection of data that are:

- **subject-oriented**: it focuses on specific subjects, such as customers, making interpretation easier;
- **integrated**: refers to the DWH's ability to **combine data from different sources**, allowing a unified analysis rather than dealing with disconnected datasets;
- **time-varying**: DW stores both **historical and contemporary data**, enabling **trend analysis**;
- **non-volatile**: data in the DWH is not updated or deleted as part of normal transactions, allowing **long-term data retention**.

The combination of these characteristics makes DWH an ideal tool for **decision making processes** and to store and analyze large amounts of data. However, the creation phase is a long and costly process, and integrating a DW often exposes data quality issues that need to be addressed.

Describe the main differences between data integration and data warehousing

When dealing with multiple data sources, there are two main approaches: a **lazy approach (data integration)** and an **edger/proactive approach (data warehousing)**.

- **Data integration**: a query-driven method where the integration system is **directly interconnected with clients**. It consists in two main phases:
    1. **Record linkage**
    2. **Data fusion**

  Data from different sources is combined only upon request, requiring specific queries. This means that data is **integrated and transformed at query time**, which can be inefficient for complex queries, as it may require on-the-fly processing of large data volumes. Additionally, **only real-time data** is considered, without storing historical data, making **trend analysis over time impossible**. The direct connection to the client also increases the system's workload.
- **Data warehousing**: **pre-integrated** and **pre-processed** data from multiple sources are stored. Since data is integrated before use, it is **already structured and organized**, making it easier to analyze and interpret. A DW is specifically designed to support complex queries and **historical analyses**, allowing for trend analysis over time. Furthermore, it is **not directly connected to the client**, meaning it can handle large data volumes without increasing the system workload.

Describe the different types of DW architectures

- **Single tier**: all the components are contained within a **single server**. It's the most basic type and it's suitable for **small businesses** with simple data.
- **Two tier**: DW is split into the **front-end** tier and the **back-end** tier. Front end tier contains **user interface** and the **reporting tools**, while back end tier contains the **database** and the **ETL tools**.
- **Three tier**: DW is divided into front-end tier, **middle** tier and back-end tier. Front-end tier contains user interface and the reporting tools, **middle tier contains the ETL tools** and the back-end tier contains the database. It is **more scalable and flexible** than the two-tier architecture.
- **Hub-and-spoken**: DW is organized around a **central hub** with **satellites data marts** connected to the hub. The hub contains the original copy of the data, enabling **decentralized data management**, where each department refers to its own data mart.
- **Centralized DW**: stores all data in a **single central location**. This is primarily a **logical approach** rather than a structural distinction.

In DWs, it is common to perform integration and quality processes on data sources, producing **reconciled data**. A major advantage of the two tier architecture is the clear **separation** between the **extraction phase** and the **feed phase**.
Data marts are a **subset or aggregation** of data from DW, containing relevant data for a **specific business area**. Dependent data marts are derived directly from the DW, reducing size and increasing efficiency. In contrast, independent data marts receive data directly from sources, bypassing a primary DW. This simplifies the design phase but can lead to conflicts when multiple independent data marts coexist.
To address these issues, two solutions exist: **"Data Mart Bus"**, which integrates data marts logically using **common dimensions**, and **"Federations"**, which perform both **logical and physical integration** between data marts.

Multidimensional model - OLAP approach

OLAP (Online Analytical Processing) supports multidimensional data analysis, requiring a multidimensional thinking approach. It is organized into **work sessions** and is designed for **non-IT users**, facilitating **data exploration across multiple perspectives**. An OLAP session consists of a **navigation path** that reflects **process-oriented analysis** on one or more facts. Query results are often reused in subsequent queries, typically presented in a tabular format, highlighting different dimensions. OLAP navigation is based on the multidimensional model.
In the **multidimensional model**, data is represented in **cubes**, where each **cell** contains numerical measures identifying **facts**, **axes** represent different **dimensions** of interest, and **hierarchies** between attributes enable **data aggregation**.
OLAP operations allow for **dynamic data exploration**. **Slice** fixes a value for one dimension, creating a sub-cube, while **dice** selects a subset of dimensions to form a new sub-cube. Aggregation methods such as **roll-up** increase aggregation by removing detail levels from the hierarchy, while **drill-down** decreases aggregation by adding more granular levels. **Pivoting** changes the representation format, and **drill-across** allows for joining multiple data cubes to enhance analytical capabilities.
By combining these approaches, DW provides a flexible, structured, and powerful environment for querying, analyzing, and extracting insights from large datasets.

Dimensional Fact Model (DFM) and dynamic dimensions

The Dimensional Fact Model (DFM) is a **graphical model** used for **designing data marts**. The DFM is designed to support **conceptual design**, provide an intuitive query environment, facilitate dialogue between designers and users for refinement, create a foundation for logical modeling, and offer comprehensive documentation. The DFM consists of a **set of fact schemas** that model the key elements of a data mart. Its basic constructs include:

- a **fact**, which represents a concept of interest for decision-making and has dynamic aspects that evolve over time (e.g., a customer making a purchase);
- a **measure**, which is a numerical property of a fact (e.g., sales measured by a receipt);
- a **dimension**, which is a finite-domain property that defines the **coordinates of analysis** (e.g., product, store, date), with facts describing a **many-to-many relationship** between dimensions;
- **dimensional attributes**, which provide **additional descriptions** for dimensions;
- a **hierarchy**, which is a **directed tree** where nodes represent dimensional attributes and edges model many-to-one associations between pairs of dimensional attributes (e.g., Store → City → Region → Country).

Example: Fact → Sale, Measure → Quantity sold, Dimension → Store.

Advanced constructs in DFM include:

- a **descriptive attribute**, which contains additional information about a dimensional attribute within a hierarchy, is linked via a one-to-one relationship, and is **not used for aggregation** because it has **continuous values**;
- a **cross-dimensional attribute**, which is a dimensional or descriptive attribute whose value depends on the **combination of two or more dimensional attributes** (e.g., a day may be classified as a weekday or holiday based on the city/region);
- **convergence**, which occurs when two dimensional attributes are connected by multiple directed paths, provided each still represents a functional dependency (e.g., Store → City → Region → Country or Store → District → Region → Country);
- **shared hierarchies**, which refer to **replicated portions of a hierarchy** within the schema;
- a **multi-arc model**, which represents a many-to-many relationship between two dimensional attributes;
- an **incomplete hierarchy**, which is a hierarchy where one or more aggregation levels are missing for some instances (e.g., Valle d'Aosta in Italy has only one province).

The DFM provides a structured way to design and represent data marts, optimizing query efficiency and analytical capabilities within a DW environment.

Logical model - Star/snowflake schema

To convert a Dimensional Fact Model (DFM) into a logical model, several steps are required, starting from the conceptual schema and ending with the production of the logical schema for the data mart. During the design phase, certain principles can be applied to optimize the schema for specific uses, often involving **redundancy** and **denormalizing relationships** to enhance query performance. This results in a **logical schema** that serves as the foundation for **data mart implementation**.
The key steps in logical design include:

- choosing the logical schema
- translating the conceptual schema
- selecting dynamic hierarchies
- optimizing the design.

The logical schema can be structured as either a **star schema** or a **snowflake schema**.

- The **star schema** consists of a **single fact table surrounded by multiple dimension tables**. The fact table stores numerical values for analysis, while the dimension tables store categories or attributes that organize the data. This structure **introduces data redundancy** but **simplifies query execution**.
- The **snowflake schema** is a variation of the star schema that further **normalizes dimension tables by splitting them into sub-tables**, which then link back to the primary dimension table. This structure allows for **greater granularity** and **more efficient queries**, but at the cost of **increased complexity** and **reduced performance**.

To translate a DFM into a star schema, a fact table is created containing all measures and descriptive attributes directly linked to the facts.
For each hierarchy, a dimension table is created, containing all relevant attributes. Additionally, advanced constructs must be managed within the schema.

There is debate over the use of snowflake schemas, as they contradict the core philosophy of Data Warehouses (DW). **The additional "beautification" can make them harder to maintain**, though they offer **significant space savings**. However, the trade-off between storage efficiency and performance cost must be carefully considered.

DATA PREPARATION

Describe, using examples, the differences between data integration and data enrichment of a dataset, enlightening common aspects and the different scenarios

- **Data integration** is the process of **combining data from multiple sources** into a single unified view. This involves extracting data from different sources, transforming it into a **common format**, and loading it into a **central repository**, such as a data warehouse. The data integration process consists of **two key phases**:
  - **record linkage**, which identifies sets of records that refer to the same real-world object
  - **data fusion**, which selects a single representative record from the identified set.
- **Data enrichment** is the process of **adding context** or **additional information** to data. This can involve **incorporating external sources** to provide more context or applying algorithms and machine learning to extract deeper insights from the data.

An example of data integration is creating a 360-degree customer view by combining information from various enterprise systems, including web traffic logs, marketing software, customer-facing applications, sales and customer support systems, and even third-party partners. These disparate data sources must be unified to support analytics and operational decision-making.
Regarding data enrichment, an example can be found in the retail sector, where companies enhance customer data to create personalized and targeted promotions for each individual customer.

Heterogeneity and homogeneous models

A **homogeneous model** is built using data from a **single source** or domain. The data used is assumed to be similar in structure and format, and the model is designed to operate on a specific type of data.
A **heterogeneous model** is built using data from **multiple sources**, which may have different structures or formats. These models are **more robust and flexible** than homogeneous models because they can **handle a wider range of input data**. However, they are also **more complex** to construct and maintain.
In the context of data integration, **heterogeneity** refers to **differences in structure, format, and content across different data sources**. Heterogeneity can be classified into:

- **Name heterogeneity**: Differences in naming conventions between sources. This includes **synonyms** (different names for the same concept) and **homonyms** (the same name used for different concepts).
- **Type heterogeneity**: Differences in data types. The same concept may be represented using **two different conceptual structures** in separate schemas.
- **Model heterogeneity**: Differences in data models or frameworks used by different sources to represent information.

Managing heterogeneous data is a critical challenge in data integration, requiring **standardization, transformation, and reconciliation** to ensure consistency and usability across different systems.

Schema integration methodology

Schema integration consists in three phases:

1. **Pre-integration**: takes n input schemas from different sources and outputs homogenized schemas, using **model transformations** and **reverse engineering** to prepare the data for integration. It can be of two types: **binary** or **n-ary**.

    a. **Binary integration** combines **two data sources** into a single integrated dataset and can be either **composed** (concatenating datasets with **similar formats**) or **balanced** (**selecting representative subsets from both datasets**).
    b. **N-ary integration** combines **three or more sources** into a single dataset, either in **one step** (if the formats are similar) or **iteratively** (adding one structure at a time).
2. **Correspondences investigation**: takes n input schemas and returns schemas with **detected correspondences** using techniques to **identify relationships** between datasets. It determines **how** data from different sources can be integrated and combined, considering **semantic relativity**, meaning that different sources may use different schemas or structures to organize and represent data. A **correspondence schema** is created to compare structures and identify **discrepancies**, which can be resolved manually or automatically.
3. **Schema integration and mapping generation**: takes n schemas and correspondences and produces an **integrated schema** along with **mapping rules** between the integrated schema and the input schemas. Integrating data from multiple sources often leads to **conflicts**, which require transformation rules to define how data should be adjusted. A **classification of conflict types** and corresponding integration rules is essential.

Different types of conflicts include **classification conflicts** (discrepancies in attribute categorization or grouping), **descriptive conflicts** (differences in attribute descriptions or definitions), **structural conflicts** (differences in attribute formats or structures), **fragmentation conflicts** (differences in data segmentation, e.g., temporal partitioning), and **instance-level conflicts** (discrepancies in specific values or attribute instances, such as attribute conflicts or key conflicts). Instance-level conflicts can be handled during design or at query time using resolution functions.
Integration rules define how data from different sources should be transformed during integration. These rules establish **mappings** between the integrated schema and source schemas, which is crucial for conflict resolution. They can involve renaming attributes, data type conversion, or other standardization processes to ensure consistency and usability in the final integrated dataset.

Differences between integration of two or more tables

Integration can occur between two tables or across multiple data sources. In the first case, it involves merging two sources into a single table while applying **deduplication**; in the second case, it requires combining multiple sources into a table by creating relationships, often using **normalization**.
Deduplication typically refers to **removing duplicate records**. This process begins with a data quality assessment, followed by data fusion to consolidate the records.
To integrate multiple tables, it is essential to improve data quality through a data quality improvement process. This includes various techniques: data cleansing (identifying and correcting errors), data standardization (converting data into a common format), data enrichment (adding more context to data), and data deduplication (identifying and removing duplicate records).
Once data quality has been improved, the next step is **record linkage**, which involves identifying records referring to the same entity. This process can be challenging and employs various techniques:

- **Empirical**: based on **statistical analysis**, assuming that records referring to the same entity will have similar values.
- **Probabilistic**: uses statistical models to estimate the probability that two records correspond to the same entity. The process includes **normalization** (restructuring records into a common format), followed by a **distance function** and a **threshold-based decision rule**.
- **Knowledge-based**: relies on predefined rules specific to a domain or expert knowledge.

- **Mixed probabilistic and knowledge-based**: combines statistical models with domain-specific knowledge, often achieving **higher accuracy** than purely probabilistic or knowledge-based methods.

After the data fusion phase, conflicts must be managed. **Conflict resolution strategies** include:

- **Conflict ignoring**: passing on the issue or considering all possible values.
- **Conflict avoiding**: selecting trusted sources, avoiding unreliable information, or excluding conflicting data.
- **Conflict resolution**: actively deciding on the most accurate value, using **mediation techniques** or **predefined resolution rules.**

Effective data integration requires careful record linkage, conflict management, and data quality improvement to ensure a coherent and reliable dataset for analysis and decision-making.

DATA QUALITY

Data quality definition

Quality is the **degree of fitness for use**, and data quality directly impacts the decisions made based on it. Quality applies not only to records but also to attributes, following the ***"garbage in, garbage out"*** principle (poor-quality data leads to poor-quality results). **Trust** in the data source is crucial.
The trade-off in data quality lies between usefulness and faithfulness. Quality attributes refer to the ability of data to meet user needs, while dimensions represent specific characteristics of information quality (usually non-measurable). Metrics, on the other hand, include methodologies and units of measurement for assessing quality, though the number of possible metrics is difficult to define.
Data issues can include **inaccuracy (incorrect data)**, **incompleteness (data that is not fully representative),** or **inconsistency (data that does not align with other sources)**. Measurements of data quality can be **objective** (formal, precise, and independent of human perception) or **subjective** (dependent on human perception and interpretation).

Data quality dimension in the relational model

- **Accuracy** measures **how closely** a data value represents the correct real-world value. It can be defined at different levels: alphanumeric values, tuples, and relationships, but not on numerical values. The accuracy of a value **v** is determined by its **distance** from **v'**, the correct value it is supposed to represent. Since **v' is often unknown**, accuracy can be costly to measure. Accuracy can be classified into **syntactic accuracy** and **semantic accuracy**.
  A data value is not syntactically accurate when it does not exist in the reference domain of possible values. For example, in a column labeled "German cars", the value "Ferrari" is syntactically inaccurate because Ferrari is an Italian car brand. Syntactic accuracy can be measured using **string-based methods** (which apply **distance functions**, either normalized **(EDnorm)** or non-normalized **(UED)**) or **token-based methods**.
  Semantic accuracy, on the other hand, refers to the degree to which data correctly represents real-world facts. A value is not semantically accurate when it misrepresents reality. For example, if a dataset contains a column with employee names and lists "Andrea Rossi", but the actual employee's name is "Alessandro Rossi", the value "Andrea Rossi" is semantically inaccurate.
- **Completeness** can be assessed at different levels: values, tuples, attributes, relationships, tables, or objects. Completeness in tuples, attributes, and tables is typically evaluated based on **null values** and is defined under the **Closed World Assumption** (CWA), meaning that if a value is missing, it is assumed to be false. An alternative approach is the **Open World Assumption** (OWA), which introduces completeness at the object level, considering that more objects may exist than the ones represented in the dataset.
- **Time-related** data quality dimensions include **currency** and **timeliness**. **Currency** refers to **how quickly data is updated** to reflect real-world changes. A fundamental measure of currency is the **time lag** between t1 (the moment a real-world event occurs) and t2 (the moment it is recorded in the system). Currency is represented as the **difference between the time of arrival in the organization and the update time**. It can be measured through log records of arrivals and updates or approximated for data with known update frequencies. **Currency is often referred to as the last update time**.
  Timeliness, on the other hand, measures the **gap between when data becomes available and when it is needed**. Unlike currency, timeliness is **process-dependent** and is associated with

the specific moment when the data must be available for use. For example, data may have high currency (recently updated) but still be obsolete for the process that requires it.

- **Consistency** has **two meanings**. The first refers to data consistency concerning **integrity constraints** defined within the schema (e.g., a Zip code must match the correct city). The second refers to **representation consistency**, meaning that the same real-world object should be consistently represented across the database (e.g., addresses must follow the same format throughout the system).
  Consistency constraints are restrictions applied to the database to ensure it remains **coherent** and logically sound. Business rules, however, are established by the organization managing the data and define how data should be used according to business policies.

ADVANCED DATA MANAGEMENT

Differences between Big Data and Open Data

Big Data are characterized by the **5V**s: **Value, Volume, Velocity, Variety**, and **Veracity,** while **Open Data** are based on **Visibility** and **Value**. Analyzing Big Data on a single server is impossible, and the solution lies in **distributed parallel architectures**.
Parallel systems, however, come with various **challenges**, including:

- **Synchronization**: lack of coordination between processes can lead to conflicts.
- **Deadlock**: occurs when two or more processes are blocked, each waiting for the other's result, potentially creating a **circular dependency**.
- **Bandwidth**: the amount of data that can be transmitted through a channel is a limiting factor for performance.
- **Coordination**: managing synchronization and task allocation across multiple processes is crucial for system efficiency.
- **Failure**: one or more processes may fail or shut down, potentially causing a **cascading failure** that affects the entire system.

To efficiently handle Big Data, distributed systems must address these challenges through **fault tolerance, load balancing**, and **optimized data distribution**.

Describe the main architectures for data distribution

Data architecture is a set of rules, policies, standards, and models that define **what** types of data are collected and **how** they are used, stored, managed, and integrated within an organization and its database systems.
Early databases were designed to run on a single computer, where data was stored and managed in a single location (centralized database). However, the efficiency of this design depends on network connectivity, data traffic, and request load. Moreover, having a single copy of the data reduces overall system efficiency, and in case of hardware failure, all data is lost due to the lack of fault tolerance mechanisms.
To overcome these limitations, **distributed databases** were developed. These can be located on **multiple computers** in the same location or distributed across a **network of interconnected computers**, forming a **distributed system**. Distribution involves **splitting data and storing it in different locations**, enabling **parallel operations**.
Distributed databases allow applications to **access data as if it were centralized**, without needing to know where the data is physically located. Despite advantages such as higher availability, autonomy, and reliability, distributed databases have complex architectures and high implementation costs and require enhanced security measures. Another challenge is the lack of standards for converting a centralized database into a distributed one.
Distributed databases can be classified as homogeneous or heterogeneous.

- **Homogeneous distributed databases**: all sites use the **same software**, are aware of each other, and cooperate, **making the system appear as a single unified database**.
- **Heterogeneous distributed databases**: different sites may use **different software and schemas**, making query processing more complex. In some cases, **sites may ignore each other's existence** and provide only limited query-processing capabilities for users.

While distributed databases improve performance, fault tolerance, and scalability, their increased complexity and security requirements make them challenging to implement and maintain.

Architectures for query elaboration

- **HDFS**: The HDFS (**Hadoop Distributed File System**) is a distributed file system designed to **run on commodity hardware**. It is **highly fault-tolerant**, **optimized for low-cost machines**, and provides **high-speed access to data**, making it ideal for large-scale datasets.
  HDFS follows a **master/slave architecture**, where an HDFS cluster consists of a single **NameNode (master server)** that manages the file system namespace and regulates client access to files. Additionally, **DataNodes** (usually one per cluster node) handle the storage associated with each node.
  The **NameNode** is responsible for **file system namespace operations**, such as opening, closing, and renaming files and directories. It also determines **block mappings** to DataNodes and redirects client read and **write requests** to the appropriate DataNodes.
  DataNodes perform **storage-related tasks**, including creating, deleting, and replicating blocks, under the NameNode's instructions.
  HDFS also maintains a Transaction Log, which records every operation performed on a file, ensuring data integrity and system reliability.
- **MapReduce**: MapReduce is a **distributed computing engine** where each program is written in a functional programming style and executed in parallel. It is inspired by the "Mapcar" and "Reduce" functions in LISP, which respectively apply an operation to all elements of a set and combine the resulting values into a final output.
  MapReduce follows a **master-slave architecture**, where the **master node** is responsible for **managing the job queue, splitting tasks into smaller work units, and monitoring their completion or failure**, while the **slave nodes execute the assigned jobs** and send status updates back to the master node.
  Each job consists of **two main operations**: Map and Reduce.
  - The **Map phase** applies the same function to a large number of records, **transforming input data into key-value pairs** and filtering them into a new set of key-value pairs. This intermediate dataset is then passed to the framework, which performs a shuffle and sort operation, grouping elements by key.
  - The **Reduce phase** aggregates the intermediate results, combining elements with the same key and producing the final output.

  MapReduce enables efficient parallel processing of large-scale datasets, making it a core component of Big Data processing frameworks such as Apache Hadoop.
- **Hadoop**: Hadoop is a framework that supports **distributed applications** with high data access, **combining the MapReduce system for parallel and distributed computation with HDFS** for storage and file management. It is characterized by being a **scalable** and **cost-effective** storage model. As data volumes increase, so does the cost of storing data online. **Reliability** is ensured by **redundant data storage across multiple servers**, which can be distributed across different racks of computers, preventing data loss in case of a server failure by shifting processing to another server.
  Additionally, it provides a **scalable processing model**, allowing data to be loaded into Hadoop without requiring conversion into a highly structured or normalized format, making it highly flexible for handling large-scale unstructured datasets.

Usefulness of Apache Spark

Apache Spark is a **general-purpose processing engine** that, unlike traditional MapReduce systems, supports a **wide range of operations beyond just map and reduce**. It is an open-source software that supports Java, Scala, and Python. The key construct in Spark is the RDD (Resilient Distributed

Dataset), which enables **fault-tolerant parallel processing** of large-scale data. Spark supports data analytics, machine learning, graph processing, and much more. It can read and write from various data sources and allows development in multiple programming languages.
Spark consists of several specialized components, including:

- Spark Core, which serves as the execution engine;
- Spark SQL, which provides an SQL-based interface;
- Spark Streaming, which enables real-time data processing;
- Spark MLlib, a machine learning library;
- GraphX, which supports graph analytics.

One of Spark's key features is its ability to **execute tasks both locally and on a cluster of machines**, ensuring scalability and efficient workload management.
To access external data, Spark integrates with Hadoop InputFormat API. A DataFrame in Spark is a distributed data structure organized in columns, similar to a relational database table, and can be constructed in various ways. Spark also enhances Hadoop's performance, introducing Apache Spark Hadoop 2.0, which includes various optimizations and support for YARN.
Additionally, Apache Accumulo, a distributed key-value storage system, improves performance, while Apache HBase, a column-oriented distributed database, enables real-time analytics. These integrations make Spark a powerful tool for Big Data processing with high flexibility, scalability, and performance efficiency.

Data lake and its main components

A Data Lake is a storage system capable of **holding an enormous amount of data in any format** in a cost-effective manner. It is an infrastructure where the **schema and data requirements are not predefined** but are **determined at the time of querying**, following a **schema-on-read approach**. This bottom-up method is characterized by experimental observation, allowing raw data acquisition and storage without an initial predefined purpose. The data remains undefined until a query is executed, making the system **highly flexible**.
One of the main advantages of a Data Lake is its ability to **store data in multiple formats** without the need for uniformity or normalization. This allows **data ingestion** from **any information source**, regardless of structure or predefined organization. The architecture of a Data Lake typically includes several key components:

- **Data ingestion**, which collects data from **various sources**;
- **Data storage**, which **stores raw data** within the Data Lake;
- **Data processing**, which **converts** raw data into a format suitable for querying;
- **Data catalog**, which organizes **metadata**;
- **Data access**, which enables users to **retrieve data**;
- **Data governance**, which defines the **policies and procedures** for data management.

The **unstructured approach** of a Data Lake provides several operational and management benefits. It significantly expands the amount of data available to analysts and increases the variety of analytical methods that can be applied, as the data remains unprocessed. Additionally, it reduces storage and management costs by **eliminating the need for predefined data structures** in both software and hardware, allowing the use of distributed file systems to lower infrastructure costs. Moreover, Data Lakes accelerate analysis by **storing data in its natural form**, ensuring that it remains unaltered by preprocessing steps, thus enabling faster and more flexible data exploration.

Differences and analogies between data lake and data warehouse

Although they may seem similar, Data Lakes and Data Warehouses are fundamentally different in several ways, representing **opposite approaches** to Big Data management, each shaped by distinct structures and objectives.

Regarding data structure, in a **Data Lake**, information is **unstructured and unprocessed**, whereas in a **Data Warehouse**, data must **first be analyzed, structured, and formatted** to fit within a **predefined and static framework** before storage.

This results in a **flexible and scalable** approach for Data Lakes, while Data Warehouses prioritize **organized and optimized** data storage.

The data analysis process also differs. In a Data Lake, analysis occurs **on-read**, meaning data is analyzed at the time of retrieval based on the specific query requirements.

In contrast, a Data Warehouse follows an **on-write** approach, where data is pre-analyzed and structured beforehand to fit into the existing schema, making it ready for immediate use.

The **purpose of data storage** is another key distinction. **Data Lakes store raw, unprocessed data without a predefined purpose**, allowing for **exploratory and flexible use**.

In contrast, **Data Warehouses store structured data with a specific predefined purpose, ensuring that information is available only for the intended analytical objectives**.

Despite their fundamental differences, the only true point of convergence between Data Lakes and Data Warehouses lies in their **core function**: both aim to manage Big Data efficiently and provide analysts with valuable insights and relevant information. However, these two antithetical approaches are **not mutually exclusive**; they can coexist within a **hybrid data architecture**, leveraging the strengths of both models to optimize data storage, processing, and analytics.