Enterprise search is the practice of making content from multiple enterprise-type sources, such as databases and intranets, searchable to a defined audience. It is a type of vertical search that is **specifically designed to help employees find information within their organization**. Enterprise search systems typically index a wide range of data sources, including **File systems, Intranets, Document management systems, Emails, Databases, Collaboration tools, Knowledge bases, Customer relationship management (CRM) systems, Enterprise resource planning (ERP) systems.** Here are some examples of how enterprise search can be used in different industries:

- Healthcare: Enterprise search can be used to help healthcare professionals find patient records, medical research, and other relevant information.
- Financial services: Enterprise search can be used to help financial professionals find financial data, investment reports, and other relevant information.
- Manufacturing: Enterprise search can be used to help manufacturing workers find product designs, engineering drawings, and other relevant information.
- Retail: Enterprise search can be used to help retail workers find product information, customer data, and other relevant information.
- Education: Enterprise search can be used to help students and faculty find research papers, course materials, and other relevant information.

nDCG10 scores are a measure of the ranking quality of a search engine or other information retrieval system. Between 0 and 1.

Data Analyst: Focus: Answering business questions using existing data. Skills: Data cleaning, wrangling, visualization, basic statistics, SQL, communication, business acumen. Responsibilities: Collecting, cleaning, and analyzing data, building reports and dashboards, identifying trends and patterns, communicating insights to stakeholders.

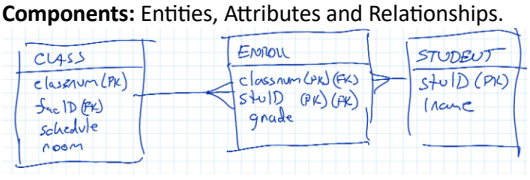
Data Engineer: Focus: Building and maintaining the infrastructure for data processing and analysis. Skills: Programming languages (Python, Java), data pipelines, databases, cloud platforms, data security. Responsibilities: Designing and building data pipelines, ETL (Extract, Transform, Load) processes, data storage solutions, ensuring data quality and security.

Data Scientist: Focus: Extracting knowledge and insights from data using advanced techniques. Skills: Machine learning, statistics, algorithms, programming languages (Python, R), data visualization, communication. Responsibilities: Building and deploying machine learning models, developing algorithms, conducting data experiments, interpreting results, communicating insights to stakeholders.

- *What type of database is this?* Databases come in a variety of genres, such as relational, key-value, columnar, document-oriented, and graph. Popular databases—including those covered in this book—can generally be grouped into one of these broad categories. You'll learn about each type and the kinds of problems for which they're best suited. We've specifically chosen databases to span these categories, including one relational database (Postgres), a key-value store (Redis), a column-oriented database (HBase), two document-oriented databases (MongoDB, CouchDB), a graph database (Neo4J), and a cloud-based database that's a difficult-to-classify hybrid (DynamoDB).
- *What was the driving force?* Databases are not created in a vacuum. They are designed to solve problems presented by real use cases. RDBMS databases arose in a world where query flexibility was more important than flexible schemas. On the other hand, column-oriented databases were built to be well suited for storing large amounts of data across several machines, while data relationships took a backseat. We'll cover use cases for each database, along with related examples.
- *How do you talk to it?* Databases often support a variety of connection options. Whenever a database has an interactive command-line interface, we'll start with that before moving on to other means. Where programming is needed, we've stuck mostly to Ruby and JavaScript, though a few other languages sneak in from time to time—such as PL/pgSQL (Postgres) and Cypher (Neo4J). At a lower level, we'll discuss protocols such as REST (CouchDB) and Thrift (HBase). In the final chapter, we present a more complex database setup tied together by a Node.js JavaScript implementation.
- *What makes it unique?* Any database will support writing data and reading it back out again. What else it does varies greatly from one database to the next. Some allow querying on arbitrary fields. Some provide indexing for rapid lookup. Some support ad hoc queries, while queries must be planned for others. Is the data schema a rigid framework enforced by the database or merely a set of guidelines to be renegotiated at will? Understanding capabilities and constraints will help you pick the right database for the job.
- *How does it perform?* How does this database function and at what cost? Does it support sharding? How about replication? Does it distribute data evenly using consistent hashing, or does it keep like data together? Is this database tuned for reading, writing, or some other operation? How much control do you have over its tuning, if any?
- *How does it scale?* Scalability is related to performance. Talking about scalability without the context of what you want to *scale to* is generally fruitless. This book will give you the background you need to ask the right questions to establish that context. While the discussion on *how* to scale each database will be intentionally light, in these pages you'll find out whether each database is geared more for horizontal scaling (MongoDB, HBase, DynamoDB), traditional vertical scaling (Postgres, Neo4J, Redis), or something in between.

Key features of NoSQL: Dynamic schema, Horizontal scalability, Document-based, key-value-based, column-based, Distributed and high availability,
Advantages of NoSQL: High Scalability, Flexibility, High availability, Scalability, Performance, Cost-effectiveness, Agility
Disadvantages: Lack of standardization, Lack of ACID compliance, Narrow focus, Open source, Lack of support for complex queries, Lack of maturity, Management challenge, GUI not available, Backup, Large document size. Flexibility and Performance.
Examples of NoSQL: **Graph Databases**: Amazon Neptune, Neo4j **Key value store:** Memcached, Redis, Coherence **Column:** Hbase, Big Table, Accumulo
Document-based: Examples – MongoDB, CouchDB, Cloudant
When should NoSQL be used: When a huge amount of data needs to be stored and retrieved. The relationship between the data you store is not that important. The data changes over time and is not structured. Support of Constraints and Joins is not required at the database level. The data is growing continuously, and you need to scale the database regularly to handle the data.
Five models used to improve search: Relevance Model, Query understanding, Document understanding, Personalization, Ranking algorithm.
MongoDB is an excellent choice for an ever-growing class of web projects with large-scale data storage requirements but very little budget to buy big-iron hardware. Thanks to its lack of structured schema, Mongo can grow and change along with your data model. If you're in a web startup with dreams of enormity or are already large with the need to scale servers horizontally, consider MongoDB.
ObjectId: JSON output of your newly inserted town contains an _id field of type ObjectId. This is akin to SERIAL incrementing a numeric primary key in PostgreSQL. The ObjectId is always 12 bytes, composed of a timestamp, client machine ID, client process ID, and a 3-byte incremented counter.
Find: db.<col_name>.find({"_id": ObjectId("")}, {<col1>:1, <col2>:1})
var population_range = {$lt: 1000000,$gt: 10000} db.towns.find({ name : /^P/, population : population_range },{ name: 1 })
db.customers.find({$and:[{'tier_and_details.active': true},{'tier_and_details.tier': "Gold" }]},{username: 1, name: 1, email: 1})

db.customers.find({$or : [{ name: /^S/ }, { birthdate : { $gte: ISODate("2000-01-01") } }]},{ username: 1, name: 1, email: 1, birthdate: 1 })
Matching exact value: { famousFor : *'food'* }, **Matching Partial value:** { famourFor: /moma/ }, **All matching values:** { famousFor : { $all : [*'food'*, *'beer'*] } }
Lack of matching values: { famousFor : { $nin : [*'food'*, *'beer'*]}}, **Elem Match:** db.countries.find({'exports.foods' : {$elemMatch : {name : 'bacon',tasty : true }}},{_id:0, name:1 })
Update: db.customers.updateOne({_id: ObjectId("64fe3342344043214785fe37")}, {$set: {state : "MI"}}), db.customers.updateMany({'tier_and_details.active': true },{$set: {is_premium: true}})
Delete: db.customers.deleteOne({_id: ObjectId("64fe3342344043214785fe37")}), db.customers.deleteMany({'tier_and_details.tier': "Silver"})
Neo4j focuses more on the relationships between values than on the commonalities among sets of values (such as collections of documents or tables of rows). In this way, it can store highly variable data in a natural and straightforward way. In Neo4j, nodes inside of graphs act like documents because they store properties, but what makes Neo4j special is that the relationship between those nodes takes center stage.
CREATE (w:Wine {name:"Prancing Wolf", style: "ice wine", vintage: 2015}) MATCH(n) RETURN n; CREATE (p:Publication {name: "Wine Expert Monthly"})
MATCH (p:Publication {name: "Wine Expert Monthly"}),(w:Wine {name: "Prancing Wolf", vintage: 2015}) CREATE (p)-[r:reported_on]->(w)
MATCH ()-[r:short_lived_relationship]-() DELETE r -> Delete RelationShip, MATCH (e:EphemeralNode) DELETE e -> Delete Node
CREATE (Salaar:Movie {title: 'Salaar Part - 1', released: 2023, tagline:'Cease Fire'})
CREATE (Rohith:Person {name: 'Rohith Anugolu', born: 1997}) CREATE (Rohith)-[:ACTED_IN {roles:['Salaar Rajamannan']}] -> (Salaar) CREATE (Rohith)-[:DIRECTED]->(Salaar)
WITH Rohith as a MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d) RETURN a,m,d LIMIT 10;
MATCH(p:Person)-[:ACTED_IN]->(m:Movie) WHERE m.released >= 1990 AND m.released <= 1999 AND m.released - p.born = 35 RETURN p,m.released;
MATCH (p1:Person)-[:ACTED_IN]->(m1:Movie)<-[:DIRECTED]-(p2:Person), (p1)-[:ACTED_IN]->(m2:Movie)<-[:DIRECTED]-(p2)
WHERE m1 <> m2 RETURN DISTINCT p1.name AS Actor, p2.name AS Director, m1.title AS FirstMovie, m2.title AS SecondMovie
CONSTRAINT: CREATE CONSTRAINT FOR (n:Movie) REQUIRE (n.title) IS UNIQUE, **INDEX**: CREATE INDEX FOR (m:Movie) ON (m.released)
DELETE ALL NODES/RELATIONSHIPS: MATCH (n) DETACH DELETE n
Applications: A* algorithm is one of the simple and efficient search algorithms that's used to find the shortest distance between two nodes in a graph. It uses weighted graphs to calculate the cost of taking each path (from current node to next node) in terms of numbers (g). The algorithm calculates how far the target node is by using a heuristic function (h). In addition to that, we use Binary Trees data structure to store the data and make the search efficient.
Redis (REmote DIctionary Service) is a simple-to-use key-value store with a sophisticated set of commands. And when it comes to speed, Redis is hard to beat. Reads are fast and writes are even faster, handling upwards of 100,000 SET operations per second by some benchmarks. Redis can also be used as a blocking queue (or stack) and a publish-subscribe system. It features configurable expiry policies, durability levels, and replication options.
You've seen **transactions** in previous databases (Postgres and Neo4j), and Redis's **MULTI** block atomic commands are a similar concept. Wrapping two operations like SET and INCR in a single block will complete either successfully or not at all. But you will never end up with a partial operation. Let's key another shortcode to a URL and increment the count all in one transaction. We begin the transaction with the MULTI command and execute it with EXEC. When using MULTI, the commands aren't executed when we define them (like Postgres transactions). Instead, they are queued and then executed in sequence. Like ROLLBACK in SQL, you can stop a transaction with the **DISCARD** command, which will clear the transaction queue. Unlike ROLLBACK, it won't revert the database; it will simply not run the transaction at all. The effect is identical, although the underlying concept is a different mechanism (transaction rollback vs. operation cancellation).
Hashes are like nested Redis objects that can take any number of key-value pairs.
MSET user:luc:name "Luc" user:luc:password s3cret | MGET user:luc:name user:luc:password | HMSET user:luc name "Luc" password s3cret | HVALS user:luc (retrieve all) | HGET user:luc password | HINCRBY | HLEN | HGETALL (get all values for key) | HSETNX
Lists contain multiple ordered values that can act both as queues (first value in, first value out) and as stacks (last value in, first value out).
RPUSH eric:wishlist 7wks gog prag | LLEN eric:wishlist | LRANGE eric:wishlist 0 -1 | LREM eric:wishlist 0 gog | LPOP eric:wishlist | RPOP | LPUSH | RPUSH | RPOPLPUSH
Sets are unordered collections with no duplicate values and are an excellent choice for performing complex operations between two or more key values, such as unions or intersections. SADD news nytimes.com pragprog.com | SMEMBERS news | SADD tech pragprog.com apple.com | SINTER news tech | SDIFF news tech | SUNION news tech | SUNIONSTORE websites news tech | SINTERSTORE | SDIFFSTORE | SMOVE | SCARD (count) | SREM news "nytimes.com" | SPOP
Database Normalization
In database management systems (DBMS), normal forms are a series of guidelines that help to ensure that the design of a database is efficient, organized, and free from data anomalies. There are several levels of normalization, each with its own set of guidelines, known as normal forms.
Important Points Regarding Normal Forms in DBMS
First Normal Form (1NF): This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries. **Second Normal Form (2NF)**: 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns. **Third Normal Form (3NF)**: 3NF builds on 2NF by requiring that all non-key attributes are independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table. **Boyce-Codd Normal Form (BCNF)**: BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key. **Fourth Normal Form (4NF)**: 4NF is a further refinement of BCNF that ensures that a table does not contain any multi-valued dependencies. **Fifth Normal Form (5NF)**: 5NF is the highest level of normalization and involves decomposing a table into smaller tables to remove data redundancy and improve data integrity. **Entity Relationship Model:** The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related. The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically. The Entity Relationship Diagram explains the relationship among the entities present in the database. ER models are used to model real-world objects like a person, a car, or a company and the relation between these real-world objects. In short, the ER Diagram is the structural format of the database.
Components: Entities, Attributes and Relationships.



DataOps is a set of practices, processes, and technologies that combine an integrated and process-oriented perspective on data with automation and methods from agile software engineering to improve the quality, speed, and collaboration in the entire data lifecycle. It's essentially about bridging the gap between data engineers and data consumers (analysts, scientists, etc.) to ensure timely and reliable delivery of valuable insights from data.
Benefits: Faster time to insights, Improved data quality, reduced costs, increased agility. **Components**: Version control, Continuous Integration/Continuous delivery (CI/CD), Monitoring and alerting, Testing and validation.

Big Query is a fully managed, serverless data warehouse from Google Cloud. It lets you store, analyze, and query large datasets, even petabytes in size, without having to manage any infrastructure yourself. Think of it as a giant storage container with powerful search capabilities, specifically designed for analyzing massive amounts of data. It is a powerful tool that can be used by businesses of all sizes to gain insights from their data. If you're looking for a way to analyze large datasets, Big Query is a great option. Here are some key features of Big Query – *Scalability, Serverless, Fast, Cost-effective, Standard SQL, Built-in machine learning.*

Column DB: A column-family database is a type of NoSQL database that stores data in a way that fundamentally differs from traditional relational databases. It focuses on organizing data by columns rather than rows, offering flexibility and scalability for handling large datasets with varied structures.

Structure: ***Columns grouped into families***: Related columns are grouped into logical units called "column families." ***Flexible schema***: Each row can have a different set of columns, unlike fixed schemas in relational databases. ***Dynamic data types***: Columns can store various data types, including text, numbers, and even complex objects. ***Time-stamped data***: Data can be automatically timestamped, providing historical context for analysis.

Column vs Row: *Advantages*: fast retrieval of small number of columns, fast writing into many optional attributes, compression (same datatype), OLAP – read small # columns for many rows, efficient big data, or in-memory processing. *Disadvantages*: slow to access in 'record' format, slow to update/delete, slow to add one new 'record' at a time, slow OLTP. **Use Cases**: Time-series data, web-scale applications, big data analytics and content management systems.

Big Table: is a fully managed, wide-column and key-value NoSQL database service offered by Google Cloud Platform. It's specifically designed for handling large analytical and operational workloads involving massive amounts of data. **Key Features:** ***Scalability***: It can handle billions of rows and thousands of columns, easily scaling to store terabytes or even petabytes of data. ***Performance***: It delivers high read and write throughput, enabling fast access to your data, even with large datasets. ***NoSQL flexibility***: It uses a key-value and wide-column store model, allowing for flexible data structures and schema-less design. ***Fully managed***: Google handles the underlying infrastructure, allowing you to focus on your data and applications. ***Cost-effective***: You only pay for the storage and queries you use, making it a cost-effective solution for large datasets. ***Durability and reliability***: Big Table offers high data durability and replication, ensuring your data is protected from loss or corruption. ***Integration with other GCP services***: Seamless integration with other Google Cloud services like Big Query and Cloud Dataflow for further analysis and processing.

Use Cases: ***Real-time analytics***: Analyze data streams from sensors, logs, and other sources in real-time. ***Operational databases***: Store and manage operational data for mission-critical applications. ***Social media data***: Store and analyze large volumes of social media data. ***IoT data***: Manage and analyze data from connected devices in the Internet of Things. ***Personalization*****:** Power personalized experiences for users based on their data. *Fraud detection*: Detect and prevent fraudulent activities in real-time.

Comparison of Big Table with Relational Databases: ***Scalability***: Can handle much larger datasets than relational databases. ***Performance***: Offers faster read and write speeds for large datasets. ***Flexibility***: More flexible data structures and schema-less design. ***Cost-effective***: Can be more cost-effective for large datasets due to its pay-as-you-go model. However, it's important to consider that BigTable is not a replacement for relational databases for all use cases. It's best suited for scenarios where scalability, performance, and flexibility are paramount, while relational databases are still preferred for applications requiring strong relational constraints and complex queries.

Sets of Orthogonal concepts for Time Series DBs: **1.Temporal Data Types:** *Instant*: something happened at an instant of time (e.g., now, which happens to be June 29, 1998, 4:06:39 P.M., when I am writing this, or sometime, perhaps much later, when you are reading this) *Interval*: a length of time (e.g., three months) *Period*: an anchored duration of time (e.g., the fall semester, August 24 through December 18, 1998) **2. Kinds of time:** *User-defined time*: an uninterpreted time value *Valid time*: when a fact was true in the modeled reality. *Transaction time*: when a fact was stored in the database. **3. Temporal statements:** *Current*: now, *sequenced*: at each instant of time, *nonsequenced*: ignoring time.

What is Time-series DB? A time series database (TSDB) is a database optimized for time-stamped or time series data. Time series data are simply measurements or events that are tracked, monitored, downsampled, and aggregated over time. This could be server metrics, application performance monitoring, network data, sensor data, events, clicks, trades in a market, and many other types of analytics data.

A time series database is built specifically for handling metrics and events or measurements that are time-stamped. A TSDB is optimized for measuring change over time. Properties that make time series data very different than other data workloads are data lifecycle management, summarization, and large range scans of many records.

Why do we need Time-series DBs? While relational databases can be used for basic time series needs, dedicated time series databases offer significant advantages in terms of data ingestion, storage, query optimization, scalability, and performance. These benefits become increasingly important as the volume, velocity, and complexity of time series data grow. It's important to choose the right tool for the job, and for complex or large-scale time series applications, dedicated time series databases are often the better choice.

1. Efficient Data Ingestion and Storage: *Time-based partitioning*: Time series databases partition data by time intervals, allowing for efficient storage, retrieval, and query execution based on time ranges. Relational databases lack this native feature, requiring manual partitioning or indexing, which can be cumbersome and inefficient. *Compression and encoding*: Time series data often exhibits temporal locality, where values change slowly over time. Time series databases leverage specialized compression techniques and encoding schemes to optimize storage and retrieval. Relational databases offer limited options in this regard.

2. Specialized Querying and Analysis: *Time-based functions*: Time series databases provide built-in functions for time-specific operations like aggregation, moving averages, and trend analysis. Relational databases require manual calculations or user-defined functions, making time-based queries cumbersome and slow. *Range queries and filtering*: Efficient retrieval of data within specific time ranges is crucial for time series analysis. Time series databases excel at range queries with optimized indexing and data partitioning, while relational databases can struggle with these due to their general-purpose nature.

3. Scalability and Performance: *High data ingest rates*: Time series databases are designed to handle high-velocity data streams from sensors, logs, and other sources. Their architecture and query optimization techniques allow for efficient ingestion without compromising performance. Relational databases can be overwhelmed by large data volumes, impacting performance and scalability. *Parallel processing and distributed systems*: Time series databases are often built with distributed architectures and parallel processing capabilities to handle massive datasets and complex queries efficiently. Relational databases, while scalable to some extent, may not be as optimized for parallel processing and distributed workloads.

Disadvantages of Time-Series DBs: 1) Time-series databases are often more complex to set up and manage than relational or NoSQL databases.2.) There are relatively few time-series databases to choose from, so you might not have as much flexibility in choosing a platform that meets your specific needs.3.) Time-series data can be very large, so you'll need enough storage capacity to accommodate your data.

Data Mart: A focused sub-set of a data warehouse, catering to the specific needs of a particular department or business unit within an organization (e.g., marketing, sales, finance). **Key Features**: ***Small and focused***: Contains data relevant to a specific department or function. ***Pre-processed and curated***: Data is pre-cleaned, transformed, and organized for immediate use. **Fast and user-friendly**: Designed for easy access and analysis by department users.

Uses: ***Departmental reporting and analysis***: Provides insights for specific business decisions. ***Performance tracking and optimization***: Monitors key metrics and identifies areas for improvement. ***Customer segmentation and targeted marketing***: Analyzes customer data for targeted campaigns.
Data Warehouse: A central repository for integrating and storing historical and current data from various sources across an organization. **Key Features:** L***arge and comprehensive***: Houses data from all departments and systems. ***Subject-oriented***: Organized by subjects like customers, products, or transactions. ***Time-oriented***: Tracks data changes over time, allowing for historical analysis. **Uses**: ***Enterprise-wide reporting and analysis***: Provides a holistic view of organizational performance. ***Data discovery and exploration***: Enables users to find patterns and trends across different data sets. ***Decision support***: Supports informed strategic decision-making based on historical data and trends.
Data Lake: A massive repository for storing all kinds of data in its raw format, structured or unstructured, without pre-processing or organization.
Key Features: ***Scalable and flexible***: Can handle any type and volume of data. ***Cost-effective***: Stores all data, regardless of immediate use. ***Open and accessible***: Data can be accessed and analyzed by various tools and technologies. **Uses**: ***Big data analytics***: Enables analysis of large and diverse datasets using various techniques. ***Data exploration and discovery***: Allows for uncovering hidden insights and patterns not readily apparent in structured data. ***Data preparation for downstream applications***: Provides a raw data source for feeding data warehouses, data marts, and other analytical tools.
Dimensional Modelling: A technique for structuring data that prioritizes ease of analysis and user comprehension, making it a favorite for business intelligence and reporting**.** Two key components of dimensional model:
1. Fact Tables: Imagine a fact table as the heart of your data warehouse. It holds the quantitative data, the "what happened" of your business. Think of it as a detailed transaction log, capturing events like sales, website visits, or customer interactions. **Purpose**: ***Store key metrics***: Fact tables house the numerical values you want to analyze, like sales amounts, website clicks, or customer satisfaction scores. ***Connect dimensions***: Each record in a fact table is linked to multiple dimension tables, providing context, and meaning to the numbers. ***Example***: Consider a fact table for online sales. Each record might contain details like order ID, product ID, customer ID, date, and sales amount. This data, on its own, is just numbers.
2. Dimension Tables: Think of dimension tables as the supporting cast, providing context and meaning to the facts. They store descriptive attributes that give your data depth and allow for meaningful analysis. Think of them like dictionaries, holding details about customers, products, locations, or time periods.
Purpose: ***Describe facts:*** Dimension tables provide descriptive information about the entities involved in the facts. For example, a customer dimension might hold details like name, location, and purchase history. ***Enable filtering and grouping***: Dimensions allow you to slice and dice your data based on specific criteria. You can analyze sales by product category, customer segment, or time. ***Enhance understanding***: Descriptive attributes in dimensions help you interpret the facts and gain deeper insights into your business. ***Example***: Returning to our sales data, a customer dimension might hold details like customer ID, name, location, and age group. These details enrich the sales data, allowing you to analyze things like which age group buys the most or which location generates the highest revenue. ***Relationship between Facts and Dimensions***: Each record in a fact table references multiple dimension tables, forming a star schema (one fact table with multiple surrounding dimensions) or a snowflake schema (multiple fact tables with shared dimensions). This interconnectedness allows you to navigate and analyze your data from various angles.
Benefits of Dimensional Modeling: ***Intuitive and easy to understand***: The structure mimics real-world business processes, making it easier for users to grasp the data and ask meaningful questions. ***Efficient querying and analysis***: The design facilitates fast and efficient retrieval of data based on specific dimensions, allowing for quick and insightful reports. ***Scalable and flexible***: can easily accommodate new data sources and dimensions without major restructuring.
Differentiate between ETL and ELT: **Technical Differences: (ETL | ELT)** ***Data processing***: Transformation happens before loading data into the target data warehouse | Transformation happens after loading data into the target data warehouse. ***Transformation tools***: ETL uses dedicated ETL tools or custom scripts to clean, transform, and integrate data from various sources | ELT leverages the processing power of the data warehouse itself for data transformation using built-in tools or external engines. ***Data format***: Data is usually converted into a structured format (e.g., relational database) before loading | Data can be loaded in its raw format (e.g., CSV, JSON) into the data warehouse. ***Target system***: The transformed data is directly loaded into the data warehouse | The raw data is loaded into the data warehouse for transformation and further analysis.
Differences in Purpose: ETL: ***Data quality and consistency***: Focuses on ensuring data quality and consistency by pre-processing and transforming data before loading. ***Reduced workload on target system***: Transformed data consumes less processing power in the data warehouse. ***Better for smaller datasets***: Efficient for smaller datasets as the entire dataset needs transformation before loading. ***Less flexible for schema changes***: Schema changes in the target system may require re-running the ETL process for existing data. **ELT**: ***Faster time to insights***: Enables faster analysis as data is directly loaded and transformed as needed. ***Scalability and flexibility***: Handles large and diverse datasets efficiently and adapts to schema changes more easily. ***Leverages data warehouse power***: Utilizes the processing capabilities of the data warehouse for efficient transformation. ***Requires more resources on target system***: Raw data loading can be resource-intensive for the data warehouse. **Choosing the Right Approach: *Data size and complexity***: ELT is more efficient for large and diverse datasets. ***Data quality requirements***: ETL offers better control over data quality. ***Transformation complexity***: ELT is more flexible for complex transformations. ***Available resources***: ETL requires additional ETL tools, while ELT relies on the data warehouse's capabilities.
Actions performed during ETL/ELT: Data transformation plays a crucial role in ETL and ELT processes, preparing raw data for analysis and consumption. Here's a list of some typical actions performed, categorized by their purpose: **1. Cleaning and Standardization: *Data cleansing***: Removing errors, inconsistencies, and duplicates from the data. ***Formatting and standardization***: Converting data to a consistent format (e.g., date format, units of measurement). ***Missing value imputation***: Filling in missing data points with appropriate values. ***Normalization***: Restructuring data to minimize redundancy and optimize storage.
2. Integration and Enrichment: ***Joining data***: Combining data from multiple sources based on shared attributes. **Aggregation and summarization**: Calculating statistical measures like sum, average, or median. ***Derived attribute creation***: Generating new attributes from existing data (e.g., calculating customer lifetime value). ***Data enrichment***: Adding additional information from external sources (e.g., geo-location data).
3. Transformation and Mapping: ***Data type conversion***: Converting data to different types (e.g., string to numeric). ***Data mapping***: Mapping data from one format or structure to another. ***Dimensionality reduction***: Reducing the number of variables for improved processing efficiency. ***Feature engineering***: Creating new features for machine learning models.
4. Validation and Verification: ***Data quality checks***: Ensuring data meets specific quality standards (e.g., completeness, accuracy, consistency). ***Schema validation***: Verifying that data conforms to the defined schema. ***Data profiling***: Analyzing data distribution and characteristics. ***Testing and debugging***: Identifying and correcting errors in the transformation process.
How APIs are used in ETL: Data Extraction, Data Transformation, Data Validation and Enrichment. **Connect to a database using API**: RESTFUL APIs, API endpoint URL, authentication credentials (API Key), specific API calls for desired data extraction and manipulation. **Benefits**: Reduced development time, Scalability and flexibility, Data automation, cost-effectiveness. **Challenges**: Complexity, Security, Data quality.
Text mining (also referred to as text analytics) is an artificial intelligence (AI) technology that uses natural language processing (NLP) to transform the free(unstructured) text in documents and databases into normalized, structured data suitable for analysis or to drive machine learning (ML) algorithms. Techniques: Info Retrieval, Tokenization (find the words), Stopping (use only useful words), stemming (analyzing).