Lesson 5.2: Database Taxonomy

UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

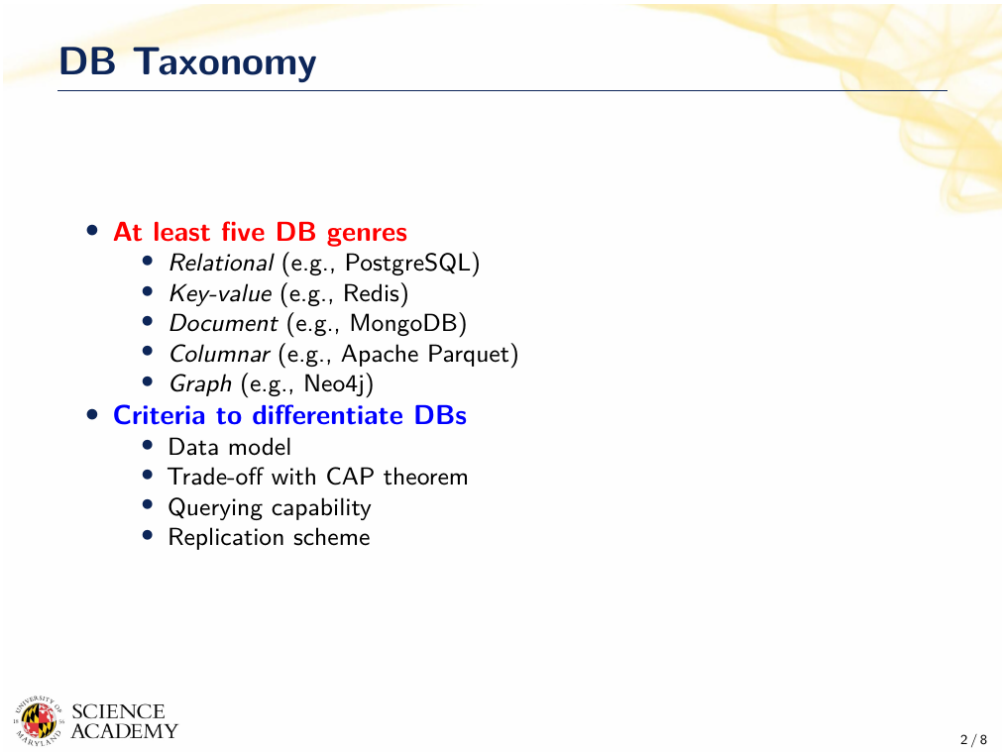
Lesson 5.2: Database Taxonomy

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu
- **References**:
  - Online tutorials
  - Silbershatz: Chap 10.2
  - Seven Databases in Seven Weeks, 2e

SCIENCE ACADEMY

1 / 8

2 / 8: DB Taxonomy

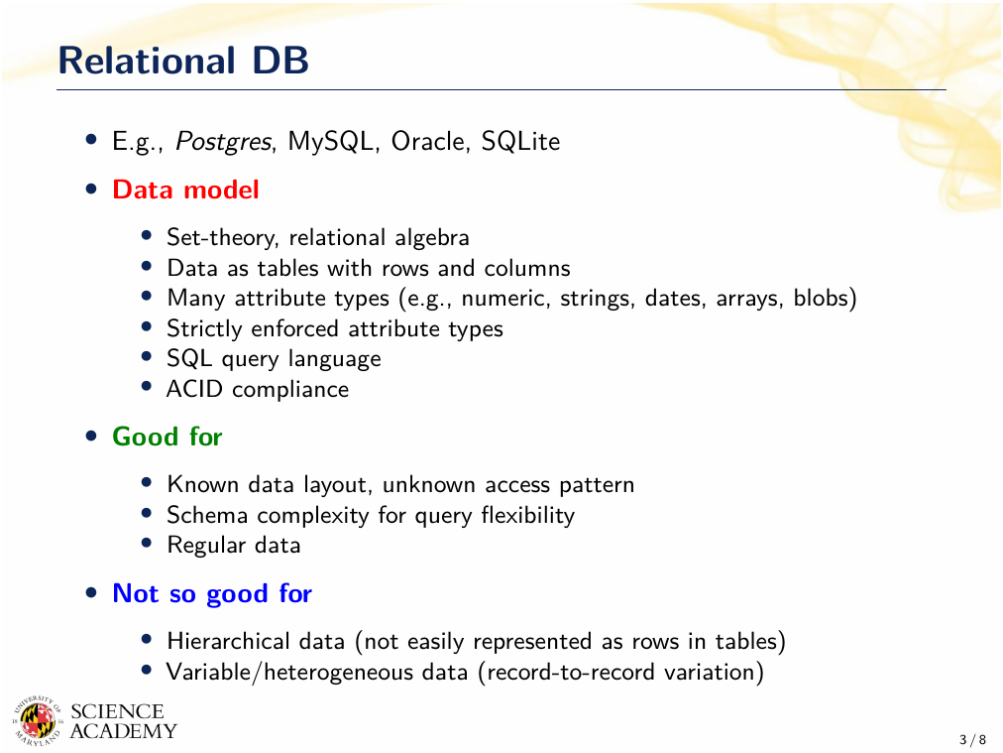


- **DB Taxonomy**
  - **At least five DB genres**
    - ***Relational* (e.g., PostgreSQL)**: These databases organize data into tables with rows and columns, using a structured query language (SQL) for defining and manipulating data. They are great for complex queries and transactions.
    - ***Key-value* (e.g., Redis)**: This type of database stores data as a collection of key-value pairs. It's simple and fast, making it ideal for caching and real-time applications.
    - ***Document* (e.g., MongoDB)**: Document databases store data in JSON-like formats, allowing for flexible and hierarchical data structures. They are well-suited for applications with varying data types.
    - ***Columnar* (e.g., Apache Parquet)**: These databases store data in columns rather than rows, optimizing for read-heavy operations and analytical queries. They are often used in big data and data warehousing.
    - ***Graph* (e.g., Neo4j)**: Graph databases use nodes, edges, and properties to represent and store data, making them perfect for applications that involve complex relationships, like social networks.
  - **Criteria to differentiate DBs**
    - **Data model**: This refers to how data is structured and stored, which can significantly impact the database's performance and suitability for different tasks.
    - **Trade-off with CAP theorem**: The CAP theorem states that a distributed database can only guarantee two out of three properties: Consistency, Availability, and Partition tolerance. Different databases prioritize these properties differently.
    - **Querying capability**: This involves the complexity and flexibility of the queries

that can be performed, which varies across different database types.

* **Replication scheme**: This refers to how data is copied and maintained across multiple locations, affecting the database's reliability and performance.

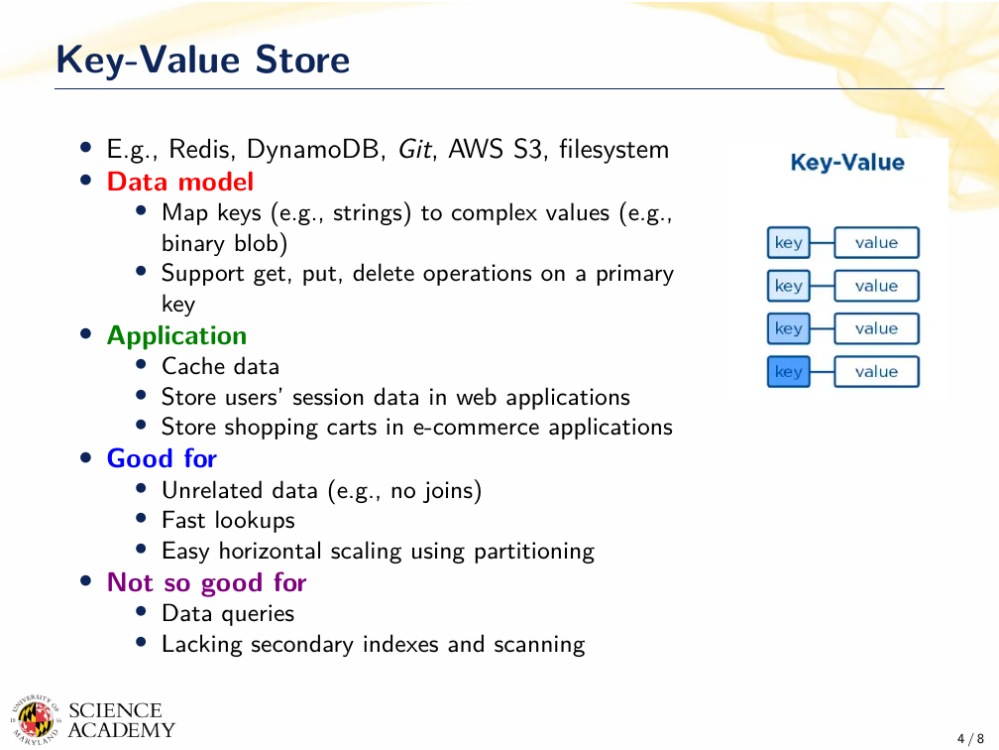
3 / 8: Relational DB



- **Relational DB**
  - Examples of relational databases include *Postgres*, MySQL, Oracle, and SQLite. These are popular systems used to store and manage data in a structured way.
- **Data model**
  - Relational databases are based on *set-theory* and *relational algebra*. This means they use mathematical concepts to organize and manipulate data.
  - Data is organized into tables, which are like spreadsheets with rows and columns. Each row represents a record, and each column represents an attribute of the data.
  - There are many types of attributes you can use, such as numbers, text, dates, and even more complex types like arrays and blobs (binary large objects).
  - Attribute types are strictly enforced, meaning that each column in a table must contain data of a specific type.
  - SQL (Structured Query Language) is used to query and manipulate the data. It's a powerful language that allows you to perform complex operations on the data.
  - Relational databases are ACID compliant, which stands for Atomicity, Consistency, Isolation, and Durability. This ensures that transactions are processed reliably.
- **Good for**
  - Relational databases are ideal when you know the structure of your data but not necessarily how it will be accessed. This flexibility allows for complex queries.
  - They are great for handling complex schemas, which means you can have intricate rela-

tionships between different tables and still query them efficiently.
  - They work well with regular data, where the structure doesn't change much over time.
- **Not so good for**
  - Relational databases struggle with hierarchical data, which is data that naturally forms a tree-like structure. This type of data doesn't fit well into tables.
  - They are not ideal for variable or heterogeneous data, where each record might have a different structure. This can make it difficult to fit such data into a fixed schema.

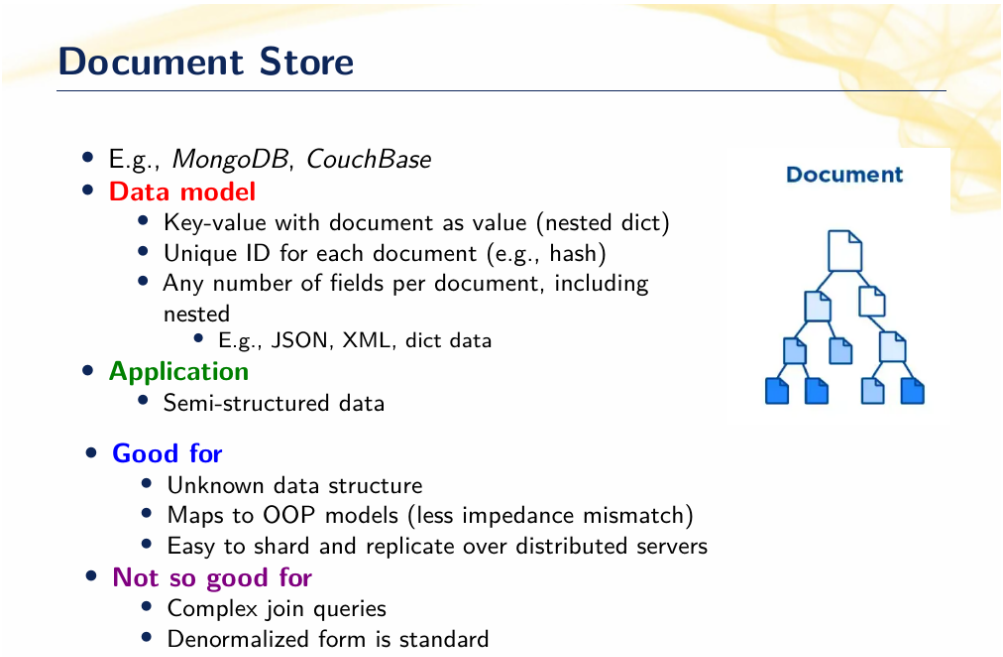
4 / 8: Key-Value Store



- **Key-Value Store Examples**
  - Examples include *Redis*, *DynamoDB*, *Git*, *AWS S3*, and traditional filesystems. These are systems designed to store, retrieve, and manage data using a simple key-value pair model.
- **Data Model**
  - The data model involves mapping *keys* (like strings) to *complex values* (such as binary blobs). This means each piece of data is stored with a unique identifier (the key), and the data itself can be any form of complex data.
  - Operations supported include *get* (retrieve data), *put* (store data), and *delete* (remove data) using the primary key.
- **Applications**
  - Key-value stores are often used to *cache data*, which helps in speeding up data retrieval.
  - They are used to store *users' session data* in web applications, ensuring quick access and updates.
  - In e-commerce applications, they can store *shopping carts*, allowing for fast retrieval and updates as users add or remove items.

- **Good for**
  - They excel in handling *unrelated data*, where there is no need for complex relationships or joins between data.
  - They provide *fast lookups* due to the simplicity of the key-value model.
  - They allow for *easy horizontal scaling* through partitioning, which means they can handle large amounts of data by distributing it across multiple servers.
- **Not so good for**
  - They are not ideal for *data queries* that require complex searching or filtering.
  - They lack *secondary indexes* and the ability to perform *scanning* operations, which limits their use in scenarios where such features are necessary.

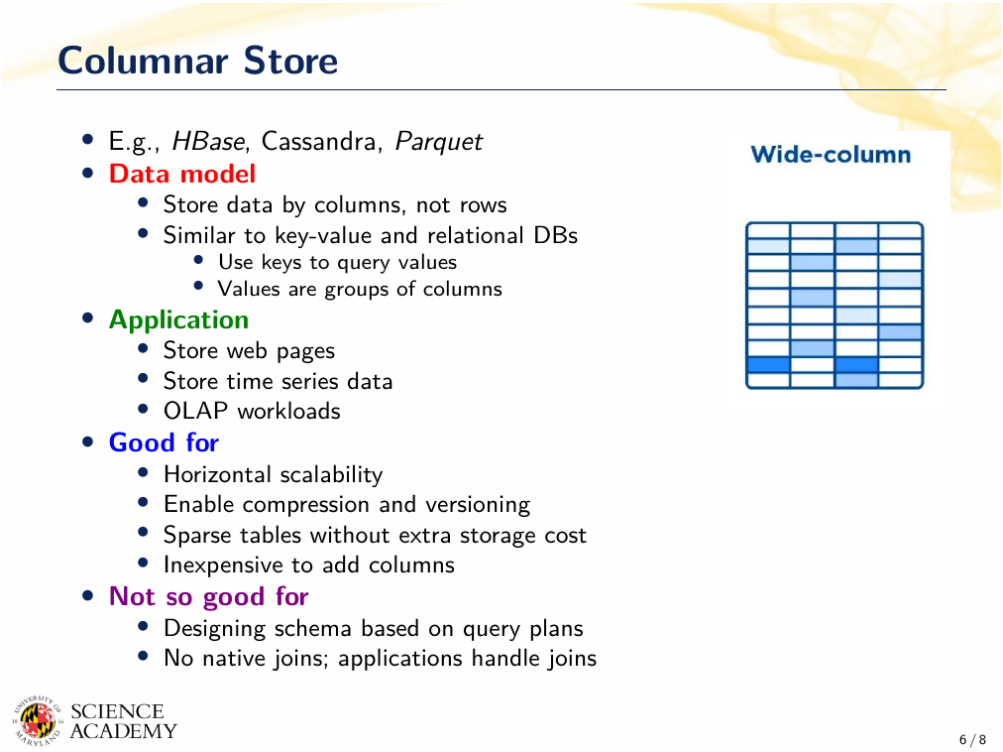
5 / 8: Document Store



- **Document Store**: This slide introduces the concept of a document store, which is a type of database designed to store, retrieve, and manage document-oriented information. Examples include *MongoDB* and *CouchBase*.
- **Data model**:
  - The data model in document stores is based on a key-value structure where the *document* acts as the value. This document is typically a nested dictionary, which means it can contain other dictionaries within it.
  - Each document is assigned a unique identifier, often a hash, which helps in efficiently retrieving and managing the documents.
  - Documents can have any number of fields, and these fields can be nested, allowing for complex data structures. Common formats for these documents include JSON, XML, and dictionary data structures.

- **Application**:
  - Document stores are particularly useful for handling *semi-structured data*, which doesn't fit neatly into traditional table structures.
- **Good for**:
  - They are ideal when the data structure is unknown or likely to change, as they offer flexibility in how data is stored.
  - Document stores align well with object-oriented programming (OOP) models, reducing the complexity of translating between the database and application code (known as impedance mismatch).
  - They are designed to be easily distributed across multiple servers, making them scalable and reliable for large-scale applications.
- **Not so good for**:
  - Document stores are not optimized for complex join queries, which are common in relational databases.
  - Data is typically stored in a denormalized form, meaning that data redundancy is common, which can lead to inefficiencies in certain scenarios.

6 / 8: Columnar Store

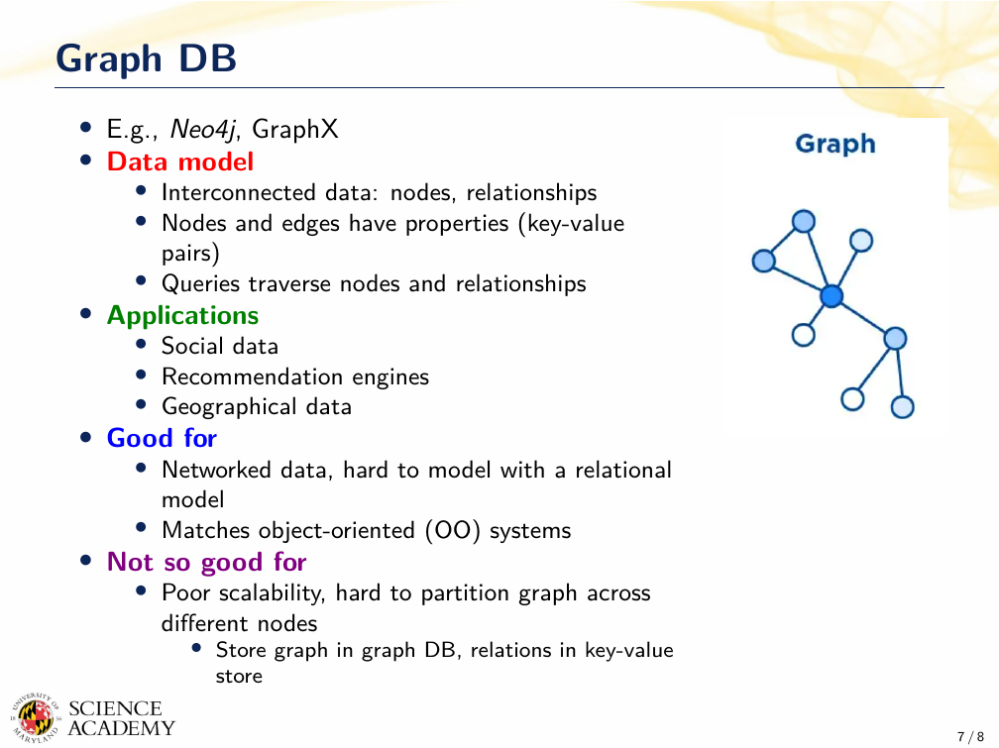


- **Columnar Store Examples**
  - Examples include *HBase*, Cassandra, and *Parquet*. These are popular technologies used to store and manage large datasets efficiently by organizing data in columns rather than rows.
- **Data Model**

  - Columnar stores organize data by columns instead of rows. This is different from traditional databases that store data in rows.
  - They are similar to key-value and relational databases in that they use keys to access data. However, the values retrieved are groups of columns, which can be more efficient for certain types of queries.
- **Application**
  - Columnar stores are well-suited for storing web pages and time series data, which often involve large datasets with many columns.
  - They are also ideal for OLAP (Online Analytical Processing) workloads, which require fast query performance on large datasets.
- **Good for**
  - These systems are designed for horizontal scalability, meaning they can easily expand by adding more servers.
  - They support data compression and versioning, which can save storage space and track changes over time.
  - Columnar stores handle sparse tables efficiently, avoiding extra storage costs for empty or null values.
  - Adding new columns is inexpensive, making it easy to adapt to changing data requirements.
- **Not so good for**
  - Designing schemas based on query plans can be challenging because columnar stores are optimized for reading rather than writing.
  - They do not support native joins, so applications must handle joins, which can complicate application development and reduce performance.

The image on the right likely illustrates the concept of columnar storage, showing how data is organized by columns rather than rows, which can help visualize the differences from traditional row-based storage systems.

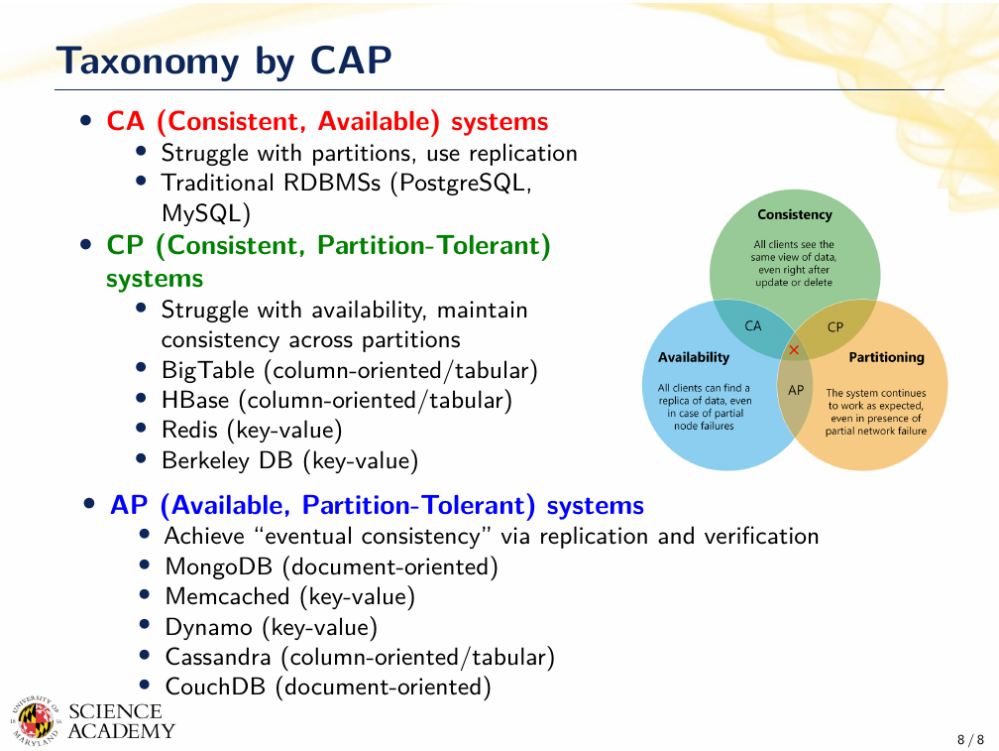
7 / 8: Graph DB



- **Graph DB Examples**
  - *Neo4j* and GraphX are popular examples of graph databases. These tools are designed to handle data that is naturally interconnected, like social networks or recommendation systems.
- **Data Model**
  - Graph databases use a model that consists of *nodes* and *relationships*. Nodes represent entities, while relationships connect these entities. Both nodes and edges (relationships) can have properties, which are essentially key-value pairs that store additional information.
  - Queries in graph databases are designed to traverse these nodes and relationships, making it easy to explore complex networks of data.
- **Applications**
  - Graph databases are particularly useful for managing *social data*, where relationships between users are crucial.
  - They are also used in *recommendation engines*, which rely on understanding connections between users and products.
  - *Geographical data* can benefit from graph databases due to the natural way they handle spatial relationships.
- **Good for**
  - Graph databases excel at handling *networked data*, which can be challenging to represent using traditional relational databases.
  - They align well with *object-oriented (OO) systems*, as both use a similar approach to modeling data.
- **Not so good for**

- One downside is their *poor scalability*. It can be difficult to partition a graph across multiple nodes, which can limit performance as the dataset grows.
- A common workaround is to store the graph structure in a graph database while using a key-value store for relationships, but this can add complexity.

8 / 8: Taxonomy by CAP



- **CA (Consistent, Available) systems**
  - These systems are designed to ensure that data is always consistent and available, but they face challenges when network partitions occur. A network partition is when there is a break in communication between different parts of a system. To handle this, CA systems often use replication, which means they keep multiple copies of data to ensure availability and consistency. Traditional relational database management systems (RDBMSs) like PostgreSQL and MySQL are examples of CA systems. They are well-suited for applications where data consistency and availability are critical, but they may not perform well in distributed environments where partitions are common.
- **CP (Consistent, Partition-Tolerant) systems**
  - These systems prioritize consistency and can handle network partitions, but they may struggle with availability. This means that during a partition, the system ensures that all nodes have the same data, but some parts of the system might not be accessible. Examples include BigTable and HBase, which are column-oriented databases, and Redis and Berkeley DB, which are key-value stores. These systems are ideal for applications where data consistency is more important than availability, such as financial transactions.
- **AP (Available, Partition-Tolerant) systems**
  - AP systems focus on being available and partition-tolerant, but they achieve only *eventual consistency*. This means that while data might not be immediately consistent

across all nodes, it will become consistent over time through replication and verification processes. Examples include MongoDB and CouchDB, which are document-oriented databases, and Memcached, Dynamo, and Cassandra, which are key-value or column-oriented stores. These systems are suitable for applications where availability is crucial, such as social media platforms, where temporary inconsistencies are acceptable.