**Types of NOSQL**

NoSQL databases are categorized based on their data model. Here are the main types:

### **1. Key-Value Pairs**

* **Description:** Store data as key-value pairs. Each key is unique, and its associated value can be any type of data.
* **Use Cases:** Caching, session management, user profiles.
* **Examples:** Redis, DynamoDB, Riak.

### **2. Document Stores**

* **Description:** Store data as documents, typically in JSON, BSON, or XML format. Each document contains semi-structured data with fields and values.
* **Use Cases:** Content management systems, catalogs, user data.
* **Examples:** MongoDB, CouchDB, RavenDB.

### **3. Column-Family Stores**

* **Description:** Store data in columns rather than rows, allowing efficient querying and aggregation for large datasets.
* **Use Cases:** Analytics, time-series data, event logging.
* **Examples:** Apache Cassandra, HBase, ScyllaDB.

### **4. Graph Databases**

* **Description:** Represent data as nodes (entities) and edges (relationships) in a graph structure.
* **Use Cases:** Social networks, fraud detection, recommendation engines.
* **Examples:** Neo4j, ArangoDB, Amazon Neptune.

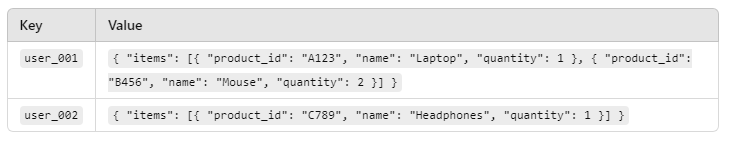
### 

### 

### 

### **Examples of the 4 Types of NoSQL Databases:**

#### **1. Key-Value Stores**

* **Example:** **Redis**
  + **Use Case:** Caching frequently accessed data for a web application.
  + **Scenario:** Storing user session data, e.g.,  
    Key: user123  
    Value: { "name": "John Doe", "lastLogin": "2024-12-03" }
* 
* **Another Example:** **Amazon DynamoDB** for scalable key-value storage.

#### **2. Document Stores**

* **Example:** **MongoDB**

**Use Case:** Storing product catalog data.

**Scenario:**json  
Copy code  
{

"productID": "123",

"name": "Wireless Mouse",

"price": 29.99,

"tags": ["electronics", "accessories"]

}

* **Another Example:** **CouchDB** for managing semi-structured data with HTTP/JSON APIs.

#### **Column-Family Stores**

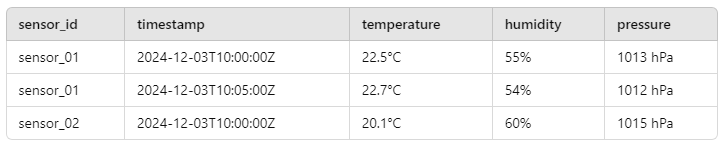
### **Use Case: Time-Series Data Storage**

Suppose we are storing sensor data for an IoT application. The data could be structured as follows:

#### **Schema in Apache Cassandra:**

* **Keyspace:** iot\_data
* **Table (Column Family):** sensor\_readings
* **Columns:**
  + sensor\_id (Primary Key)
  + timestamp (Clustering Key)
  + temperature
  + humidity
  + pressure

#### **Sample Data Representation:**

****

#### **Key Features:**

#### Data is stored by row keys (e.g., sensor\_id) and partitioned across nodes for scalability.

#### Columns are grouped and queried efficiently, especially for use cases with sequential or grouped data like time-series logs.

#### 

#### 

#### **Graph Databases**

* **Example:** **Neo4j**
  + **Use Case:** Modeling relationships in a social network.
  + **Scenario:** Nodes represent people, and edges represent relationships:
    - Node A: John
    - Node B: Jane
    - Edge: FRIENDS\_WITH
* **Another Example:** **Amazon Neptune** for managing complex, interconnected data.

**CAP Theorem**

The CAP Theorem, also known as Brewer's Theorem, is a fundamental concept in distributed systems, including NoSQL databases. It states that a distributed database can guarantee only two out of three properties at any given time:

### **CAP Properties**

1. Consistency (C):
   * Every read receives the most recent write or an error.
   * Example: If you update a record in one node, all subsequent reads across all nodes reflect that update.
2. Availability (A):
   * Every request (read/write) receives a response, even if it's not the most recent data.
   * Example: Even if part of the system fails, the database continues to operate and serve requests.
3. Partition Tolerance (P):
   * The system continues to function despite network partitions or communication breakdowns between nodes.
   * Example: If two parts of the database cluster can't communicate, the system doesn't crash.

### **Trade-offs in NoSQL**

Since network partitions (P) are unavoidable in distributed systems, NoSQL databases must choose between Consistency and Availability:

1. CP (Consistency + Partition Tolerance):
   * Guarantees consistency across nodes but might sacrifice availability during a network partition.
   * Examples:
     + HBase
     + MongoDB (in strong consistency mode)
2. AP (Availability + Partition Tolerance):
   * Ensures availability even if some nodes are unreachable, but data consistency may be temporarily compromised.
   * Examples:
     + Cassandra
     + DynamoDB
     + CouchDB
3. CA (Consistency + Availability):
   * Not achievable in distributed systems with partitions; only possible in non-distributed databases.

### **Application Scenarios**

* Consistency-Critical Applications: Banking systems, stock trading platforms, etc. (CP systems preferred).
* High-Availability Applications: Social media, online stores, IoT applications (AP systems preferred).

Understanding the CAP theorem helps in selecting the right NoSQL database for specific use cases based on business requirements.