NoSQL

NoSQL databases are a broad class of database management systems that differ from traditional relational database management systems (RDBMS) in that they do not use the standard relational database model. The term "NoSQL" originally stood for "non-SQL" to emphasize their departure from SQL-based relational databases, but it has since been interpreted to mean "**not only SQL**," indicating that some NoSQL databases support query languages that are SQL-like.

Key Characteristics of NoSQL Databases:

- **Schema-less**: NoSQL databases do not require a fixed schema before data insertion, allowing for more flexible data models that can evolve with the application's needs.
- **Scalability**: *Scalability can generally be achieved in two ways **vertical scaling** and **horizontal scaling**.*
    - ***Veertical Scaling** : Involves adding more resources (CPU, RAM, storage) to a single server or node.*
    - ***Horizontal Scaling**: Involves adding more servers or nodes to a distributed system, spreading out the data and workload across many machines rather than relying on a single one.*
- **Variety of Data Models**: NoSQL databases support a wide range of data models, including key-value, document, wide-column, and graph formats, that is unstructured data.

Types of NoSQL Databases:

1. **Key-Value Stores**: The simplest form of NoSQL databases, storing data as a collection of key-value pairs. Examples include Redis and Amazon DynamoDB.
2. **Document Stores**: These databases store data as documents, typically in JSON or BSON (Binary JSON) formats. They are schema-less, which means the structure of these documents can change over time. Examples include MongoDB and Couchbase.
3. **Wide-Column Stores**: Similar to relational databases in that they store data in tables with rows and columns, but unlike RDBMS, the names and format of the columns can vary from row to row in the same table. . Examples include Apache Cassandra and Google Bigtable.
4. **Graph Databases**: They represent and store data in terms of entities and the relationships between these entities. Examples include Neo4j and Amazon Neptune.

Use Cases:

- **Big Data and Real-Time Web Applications**: NoSQL databases are well-suited for handling large volumes of data with varying structures, or when an application requires real-time access to data.
- **Flexible Data Models**: Applications that require a flexible data model with the ability to change the data schema.
- **Scalability**: When an application needs to scale horizontally across many servers to handle large loads or large volumes of data, NoSQL databases are often more cost-effective and technically feasible than traditional RDBMS.

Advantages and Disadvantages:

- **Advantages**:
  - Scalability: Designed to horizontal scale, that is involve across many servers.
  - Flexibility: Schema-less models allow for flexible and rapid development.
  - Variety: high volume, unstructured data.
- **Disadvantages**:
  - Consistency: Many NoSQL databases sacrifice ACID (Atomicity, Consistency, Isolation, Durability)
  - Maturity: Relational databases benefit from decades of research, optimization, and tooling, while NoSQL solutions are generally newer and may lack comprehensive tools and best practices.
  - Complexity: The variety of data models can introduce complexity in selecting and managing the appropriate NoSQL database for specific use cases.

1. Document stores

1. “document" refers to the main unit of storage, which encapsulates data in formats like JSON, BSON. These databases are designed to store, retrieve, and manage document-oriented information, typically in JSON, BSON (Binary JSON), or XML formats.
2. Document stores provide a flexible schema approach, which means that the **structure of the data can change from one document to another within the same database**.
3. This flexibility makes document stores particularly well-suited for applications dealing with large volumes of **unstructured data** that may not fit neatly into the rows and columns of a traditional relational database.

Key Features of NoSQL Document Stores:

- **Schema-less Data Storage:** Documents can contain different sets and types of attributes.
- **Scalability**: Many document stores offer horizontal scalability, that is adding more servers in a distributed architecture. This makes them suitable for web-scale applications.
- **High Performance**: Document stores are optimized for fast data retrieval. Indexing on document attributes can further enhance query performance.

Example

MongoDB, Apache CouchDB, Couchbase, DocumentDB

A data base for Books in MongoDB is represented as.

```
[
  {
    "isbn": "100-100-100",
    "title": "Alchemist",
    "authors": ["Paulo Coelho"],
  },
  {
    "isbn": "100-100-101",
    "title": "The Godfather",
    "authors": ["Mario Puzo"],
    "publisher": "Penguin Books",
    "genres": ["Novel", "Psychological Fiction"],
  },
```

Above we can see two records of books , **Alchemist** and **God father** having different schema. That is the record for the book “God father“ got two more attributes (publisher and genres). This is described as **flexible schema**

Insert operation

```
db.book.insert({
    isbn: "100-100-103",
    title: "Hamlet",
    authors: ["William Shakespeare"],
    publicationYear: 1599,
    genres: [“play”, “Shakespearean tragedy”],
})
```

Query operation

```
db.book.find({title: "Hamlet"})
db.book.find({isbn: 100-100-103)}
```

Both query return same data

```
{
    "_id" : ObjectId("someObjectId"),
    "isbn": "100-100-104",
    "title": "Hamlet",
    "authors": ["William Shakespeare"],
    "publicationYear": 1599,
    "genres": ["play", "Shakespearean tragedy"]
}
```

2. Key-value stores

1. Organize data as a collection of key-value pairs, where a key serves as a unique identifier and is mapped to a corresponding value.
2. This simple data model allows for highly efficient data retrieval, storage, and management, particularly for use cases requiring fast access to large amounts of data.

Characteristics of Key-Value Stores:

Simplicity: The model is straightforward, with each key acting as a unique identifier through which its associated value can be accessed.
Performance: Key-value stores are optimized for speed, particularly for read and write operations, due to their simple data model and the ability to distribute data across multiple nodes.
Scalability: Many key-value databases are designed to scale out horizontally, making it easy to increase capacity and throughput by adding more nodes to the system.
Flexibility: Values stored can be anything from simple data types (such as strings or numbers) to more complex objects or binary data. The structure of the value is not enforced by the database, providing flexibility in what can be stored.
Schema-less: There is no fixed schema required for the data, allowing the structure of values to change without the need for database migrations.

Ecample
Redis, Amazon DynamoDB, Riak, Riak

To represent the given book records as key-value pairs suitable for a key-value store like Amazon DynamoDB, you can structure each record with its ISBN as the key and the rest of the book's information as the value.

```
{
  "100-100-100": {
    "title": "Alchemist",
    "authors": ["Paulo Coelho"],
  },
  "100-100-101": {
    "title": "The Godfather",
    "authors": ["Mario Puzo"],
    "publisher": "Penguin Books",
    "genres": ["Novel", "Psychological Fiction"],
  },
}
```

Insert operatio

```
aws dynamodb put-item \
    --table-name Books \
    --item '{
        "isbn": {"S": "100-100-103"},
        "title": {"S": "Hamlet"},
        "authors": {"L": [{"S": "William Shakespeare"}]},
        "publicationYear": {"N": "1599"},
        "genres": {"L": [{"S": "play"}, {"S": "Shakespearean tragedy"}]}
    }' \
```

Query operation

```
aws dynamodb get-item \
   --table-name Books \
   --key '{"isbn": {"S": "100-100-103"}}' \
```

This will return..

```
{
   "Item": {
      "isbn": {"S": "100-100-103"},
      "title": {"S": "Hamlet"},
      "authors": {"L": [{"S": "William Shakespeare"}]},
      "publicationYear": {"N": "1599"},
      "genres": {"L": [{"S": "play"}, {"S": "Shakespearean tragedy"}]}
   }
}
```

3. Wide-column stores

Wide-column stores are a type of NoSQL database that organizes data into tables, rows, and dynamic columns. They combine elements of both relational databases and key-value stores but are unique in their approach to column management. **Wide-column stores allow for each row to have a different set of columns**.

Key Features of Wide-Column Stores:

- **Dynamic Columns**: Unlike traditional relational databases that require a fixed schema with a predetermined set of columns, wide-column stores allow each row to have a varying number of columns. This flexibility is particularly useful for storing data with many attributes that may not be uniform across all records.
- **Efficient Storage and Retrieval**: They are optimized for queries over large datasets and can efficiently retrieve specific columns across a subset of rows.
- **Scalability**: Wide-column stores are designed to scale horizontally across many nodes, making them suitable for applications that require high performance, availability, and scalability.
- **Composite Keys**: They often support composite keys, consisting of a partition key and clustering keys, allowing for efficient data access patterns and enabling data to be stored and retrieved in sorted order.

Examples

Apache Cassandra, Google Cloud Bigtable, ScyllaDB

Cassandra encourages denormalization and duplication of data across multiple tables to optimize read performance.

Creating Schema

```
CREATE TABLE books (
   isbn TEXT PRIMARY KEY,
   title TEXT,
        authors LIST<TEXT>,
        publicationYear INT,
        publisher TEXT,
        genres LIST<TEXT>,
    );

// inseting three attributes for Book - Alcheist
INSERT INTO books (isbn, title, authors)
VALUES ('100-100-100', 'Alchemist', ['Paulo Coelho']);

// inseting three attributes for Book - Alcheist
INSERT INTO books (isbn, title, authors, publicationYear, publisher, genres)
VALUES ('100-100-101', “The God father”, ['Mario Puzo'], 1988, 'HarperCollins', ['Novel',
'Philosophical']);
```

Schema Flexibility of Cassandra

- Although Cassandra requires a schema to define tables and columns, it is more flexible compared to traditional RDBMS.
- Each row in the same table can have a different set of columns with some columns being present in one row but absent in others. Additionally,
- Cassandra supports adding new columns to existing tables without affecting existing rows

So the major difference are

1. cells can have more than one value
2. when new column is added, it will not be allocated in memory for older rows.

Modify the Table Schema

Now we want to insert a record with a new attribute “Language”

```
    For eg:
        isbn :'100-100-103',
        title: '100 years of Solitude',
        authors: ['Gabriel Garcia Marques'],
        publicationYear: 1967
        publisher: Green Books
        genres : [‘Magical Realism’]
        language: ‘spanish’
```

In Apache Cassandra, if you need to add a new attribute (like "language" in your example) to an existing table, you would typically alter the table schema to include this new column.

This will not add memory overhead to existing rows whereas in RDBMS, scema updates will add memory overhead to all exisitng records.

The process involved are

1. update existing schema using ALTER TABLE
2. insert new record

ALTER TABLE books ADD language text;

INSERT INTO books (isbn, title, authors, publicationYear, publisher, genres, language) VALUES ('100-100-103', '100 years of Solitude', ['Gabriel Garcia Marques'], 1967, 'Green Books', ['Magical Realism'], 'spanish');

Query Operation

SELECT * FROM books WHERE isbn = '100-100-103';

4. Graph stores

1. Graph stores, or graph databases, are a type of NoSQL database designed to treat relationships between data as equally important to the data itself.
2. They are optimized for managing interconnected data and for queries that traverse complex relationships with many hops in the data.
3. Graph databases excel at handling data whose relationships are as important or more important than the data itself.

Key Features of Graph Stores

- **Structure**: Graph databases use graph structures for semantic queries, with nodes, edges, and properties to represent and store data. Each node represents an entity (such as a person, business, or any other item), and each edge represents a connection or relationship between two nodes. Each node and edge can have properties associated with it.

- **Schema-less**: Similar to other NoSQL databases, graph databases are generally schema-less. This means that nodes and edges can have arbitrary and varying sets of attributes.

- **Relationship-First**: In graph databases, focuses on/ relationships . This means **they are stored at the individual record level** and can be navigated efficiently. This is in contrast to relational databases where relationships are derived by joining tables which can be computationally expensive.

Example

Neo4j, ArangoDB, Amazon Neptune, OrientDB

Designing the Graph Schema

Here's how you might design the graph model:

- **Nodes**: Each entity type (Book, Author, Publisher, Reviewer) becomes a node.
- **Relationships**: Define relationships such as `AUTHORED_BY` between Books and Authors, `PUBLISHED_BY` between Books and Publishers, and `REVIEWED_BY` between Books and Reviewers.
- **Properties**: Each node can have properties. For books, properties might include ISBN, title, publication year, and genres. Authors might have names as properties, publishers could have names and locations, and reviewers might have usernames.

Example Graph Representation

Let's take the books and their related data you mentioned and translate them into a Cypher query for Neo4j:

```
// Create Authors
CREATE (a1:Author {name: "Paulo Coelho"})
CREATE (a2:Author {name: "Mario Puzo"})
CREATE (a3:Author {name: "Gabrierl Garcia Marques"})

// Create Publishers
CREATE (p1:Publisher {name: "Penguin Books"})
CREATE (p2:Publisher {name: "Green Bools"})

// Create Books
CREATE (b1:Book {isbn: "100-100-101", title: "The Alchemist"})
CREATE (b2:Book {isbn: "100-100-102", title: "The God Father", publisher: “Penguin Books”,
genres: ['Novel', 'Psychological Fiction']})
CREATE (b3:Book {isbn: "100-100-103", title: "100 years of Solitude", publicationYear: 1967,
publisher: “Green Books”, genres: ['Novel', 'Fiction', 'Coming-of-Age']})

// Establish Relationships
CREATE (b1)-[:AUTHORED_BY]->(a1)

CREATE (b2)-[:AUTHORED_BY]->(a2)
CREATE (b2)-[:PUBLISHED_BY]->(p2)

CREATE (b3)-[:AUTHORED_BY]->(a3)
CREATE (b3)-[:PUBLISHED_BY]->(p3)
```

Query Operations

To find a book by its ISBN, you can use the following query:

```
MATCH (b:Book {isbn: "100-100-103"})
RETURN b;
```

find all books authored by "Gabriel Garcia Marques", use a query like:

```
MATCH (a:Author {name: "Gabriel Garcia Marques"})-[:AUTHORED_BY]-(b:Book)
RETURN b;
```