New-1

01 September 2024 09:46

Two-Phase Locking (2PL) protocol is a technique used in database systems to ensure that transactions are executed in a way that maintains data consistency, even when multiple transactions are happening at the same time. Let's break it down in simple words.

What is Two-Phase Locking (2PL)?

Two-Phase Locking (2PL) is a method that databases use to manage how multiple transactions access the data. It makes sure that transactions don't mess up each other's work. Imagine you and your friend are writing on the same notebook. To avoid scribbling over each other's notes, you agree on a rule: only one of you can write at a time.

How Does 2PL Work?

2PL has two phases: the **Growing Phase** and the **Shrinking Phase**.

1. **Growing Phase**: In this phase, a transaction can acquire locks but cannot release any lock. A lock is like a 'hold' on a data item that says, "I'm using this, don't change it until I'm done." So, in the growing phase, the transaction keeps getting all the locks it needs.
2. **Shrinking Phase**: Once a transaction releases its first lock, it enters the shrinking phase. Now, it cannot acquire any new locks, but it can release the locks it already holds. This phase ensures that once a transaction starts to release locks, it will not get any new locks and will eventually complete.

Why Use 2PL?

Using 2PL helps prevent problems like dirty reads, lost updates, and uncommitted data being used. These issues can occur when transactions interfere with each other. By following the two phases, 2PL ensures that transactions are isolated and data remains consistent.

Example in Simple Terms

Imagine two friends, A and B, sharing the same notebook:

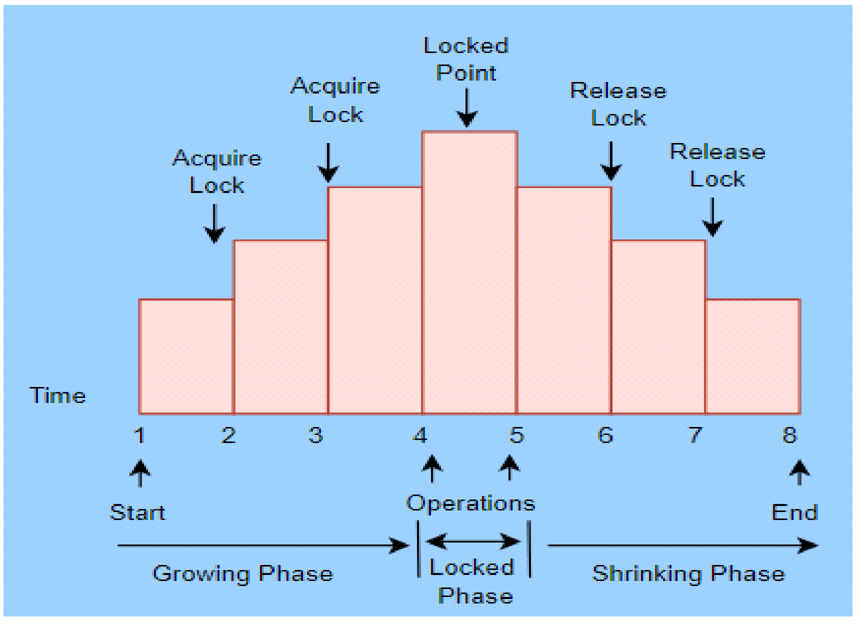
- **Growing Phase**:
  - Friend A picks up the pen (acquires lock on the pen).
  - Friend A starts writing on page 1 (acquires lock on page 1).
  - Friend A then moves to page 2 and starts writing (acquires lock on page 2).
- **Shrinking Phase**:
  - Friend A finishes writing on page 2 and puts the pen down (releases lock on page 2).
  - Friend A then stops writing on page 1 and puts the pen down (releases lock on page 1).
  - Friend A is done (cannot acquire any more locks).

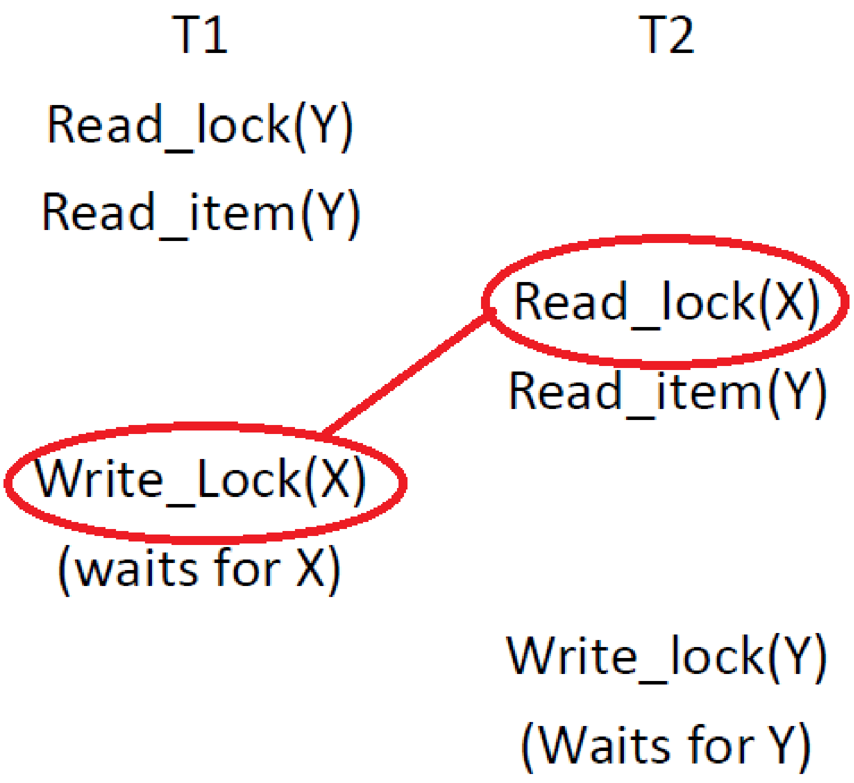
Now, Friend B can pick up the pen and start writing, following the same process.

Key Points

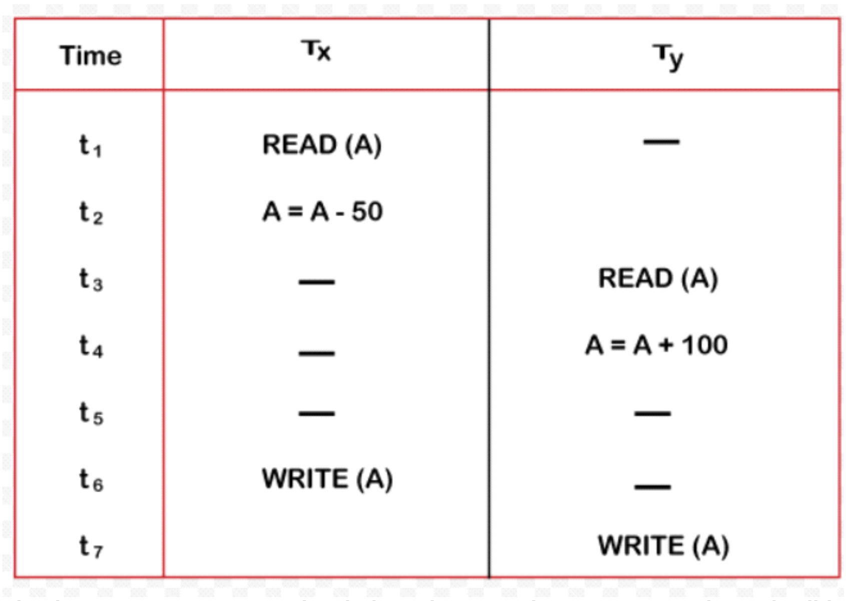
- **Growing Phase**: Acquire locks, no releasing.
- **Shrinking Phase**: Release locks, no acquiring.
- Prevents transaction conflicts and keeps data consistent.

By following these simple rules, the Two-Phase Locking protocol ensures that transactions in a database do not interfere with each other, keeping the data correct and consistent.





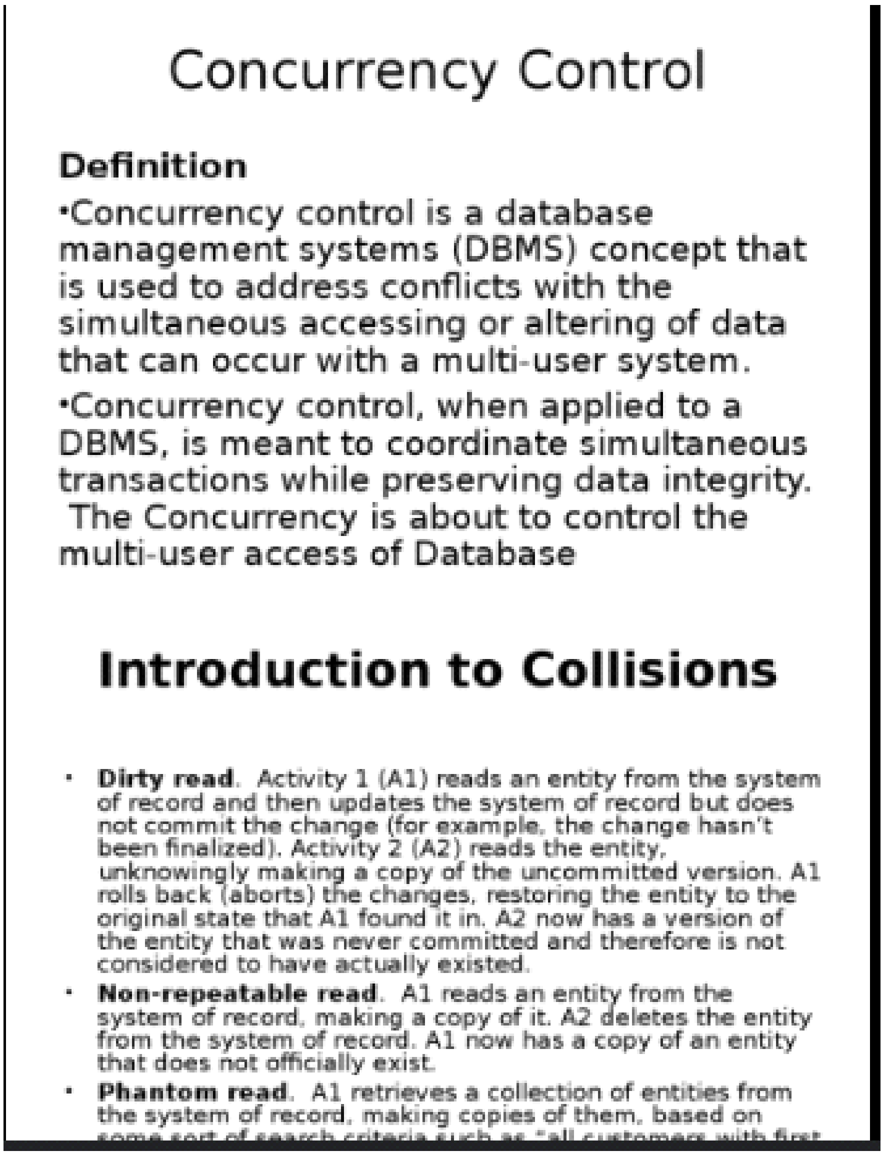
T1	T2
Read_lock(Y)	
Read_item(Y)	
	Read_lock(X)
	Read_item(Y)
Write_Lock(X)	
(waits for X)	
	Write_lock(Y)
	(Waits for Y)



Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

Timestamp ordering is a concurrency control technique that uses timestamps to order and validate transactions in a database management system (DBMS). Here's how it works:

- **Assign timestamps: When a transaction starts, it's assigned a unique timestamp.**
- **Check timestamps: The DBMS checks the timestamps for every operation.**
- **Abort and restart: If a transaction tries to access an object in a way that violates the timestamp order, the transaction is aborted and restarted.**
- **Update read timestamp: If a transaction reads an object after its write timestamp, the read timestamp is set to the transaction's timestamp.**



Concurrency Control

Definition

•Concurrency control is a database management systems (DBMS) concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system.

•Concurrency control, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity. The Concurrency is about to control the multi-user access of Database

Introduction to Collisions

- **Dirty read**. Activity 1 (A1) reads an entity from the system of record and then updates the system of record but does not commit the change (for example, the change hasn't been finalized). Activity 2 (A2) reads the entity, unknowingly making a copy of the uncommitted version. A1 rolls back (aborts) the changes, restoring the entity to the original state that A1 found it in. A2 now has a version of the entity that was never committed and therefore is not considered to have actually existed.
- **Non-repeatable read**. A1 reads an entity from the system of record, making a copy of it. A2 deletes the entity from the system of record. A1 now has a copy of an entity that does not officially exist.
- **Phantom read**. A1 retrieves a collection of entities from the system of record, making copies of them, based on some sort of search criteria such as "all customers with first

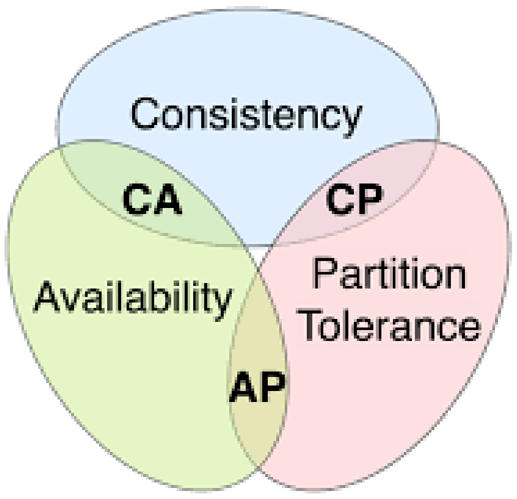
https://www.scribd.com/presentation/320810503/Concurrency-Control-in-DataBase

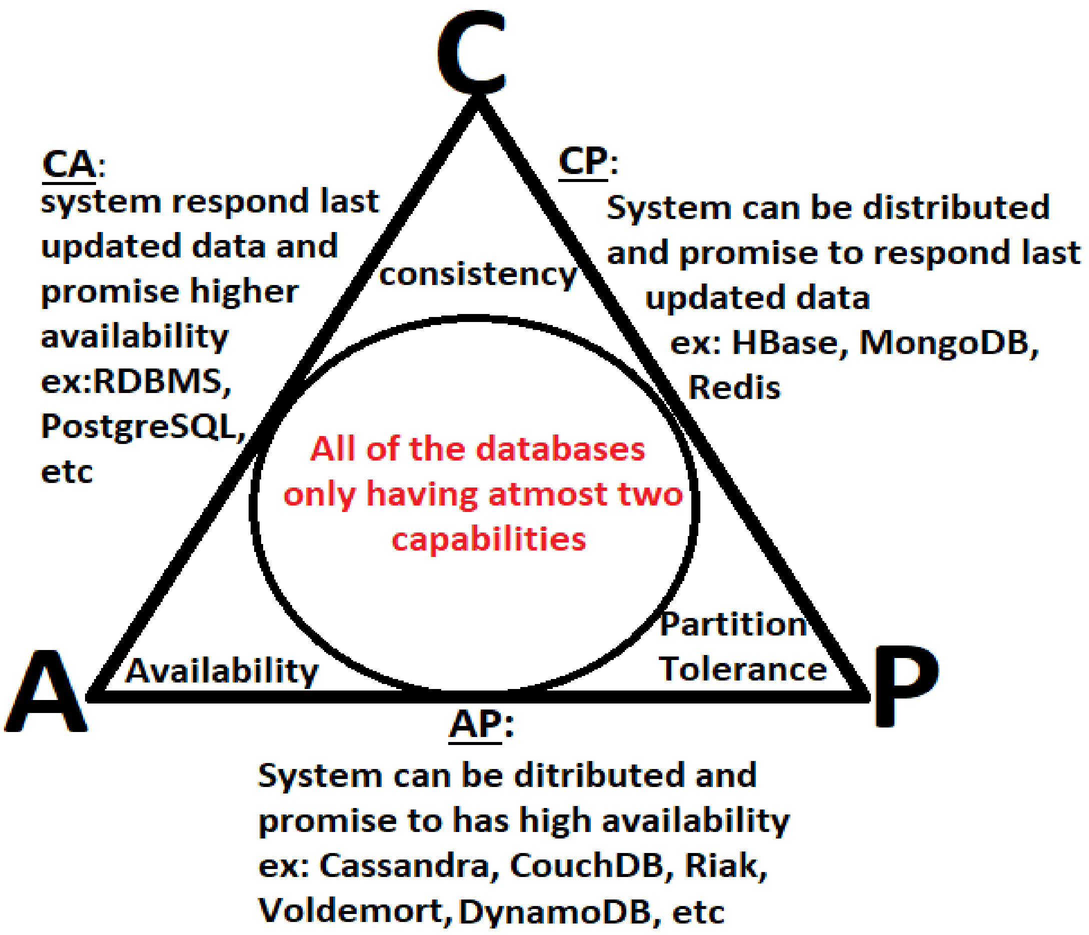
- **Dirty read**. Activity 1 (A1) reads an entity from the system of record and then updates the system of record but does not commit the change (for example, the change hasn't been finalized). Activity 2 (A2) reads the entity, unknowingly making a copy of the uncommitted version. A1 rolls back (aborts) the changes, restoring the entity to the original state that A1 found it in. A2 now has a version of the entity that was never committed and therefore is not considered to have actually existed.
- **Non-repeatable read**. A1 reads an entity from the system of record, making a copy of it. A2 deletes the entity from the system of record. A1 now has a copy of an entity that does not officially exist.

- **Non-repeatable read**. A1 reads an entity from the system of record, making a copy of it. A2 deletes the entity from the system of record. A1 now has a copy of an entity that does not officially exist.

- **Phantom read**. A1 retrieves a collection of entities from the system of record, making copies of them, based on some sort of search criteria such as "all customers with first name Bill."A2 then creates new entities, which would have met the search criteria (for example, inserts "Bill Klassen" into the database), saving them to the system of record. If A1 reapplies the search criteria it gets a different result set

What is NOSQL? Explain the CAP theorem




C
CA:
system respond last updated data and promise higher availability ex:RDBMS, PostgreSQL, etc
CP:
System can be distributed and promise to respond last updated data
ex: HBase, MongoDB, Redis
consistency
All of the databases only having atmost two capabilities
Availability
Partition Tolerance
A
P
AP:
System can be ditributed and promise to has high availability
ex: Cassandra, CouchDB, Riak, Voldemort, DynamoDB, etc

- **Consistency (C):** Every read gets the latest data.
- **Availability (A):** Every request gets a response.
- **Partition Tolerance (P):** The system stays up despite network issues.

NoSQL (Not Only SQL) refers to a variety of database systems that are not based on the traditional relational model of databases (RDBMS). Unlike SQL databases that use structured query language

(SQL) and have fixed schemas, NoSQL databases are designed to handle a wide variety of data types, including structured, semi-structured, and unstructured data. NoSQL databases are often used for big data and real-time web applications due to their flexibility, scalability, and ability to handle large volumes of data with high speed and availability.

Types of NoSQL Databases:

1. **Document-based** (e.g., MongoDB)
2. **Key-Value Stores** (e.g., Redis)
3. **Column-family Stores** (e.g., Cassandra)
4. **Graph-based** (e.g., Neo4j)

Document-based NoSQL Systems: Document-based NoSQL databases store data in documents, typically using JSON-like structures (such as BSON in MongoDB). Each document contains key-value pairs and can contain nested structures like arrays and sub-documents. These databases are schema-less, meaning each document in a collection can have a different structure, providing flexibility and scalability.
Examples: MongoDB, CouchDB, Amazon DocumentDB.
CRUD Operations in MongoDB: CRUD stands for Create, Read, Update, and Delete, which are the four basic operations for managing data in MongoDB:

1. **Create**:
   - Use the insertOne() or insertMany() methods to add a new document to a collection.

     javascript
     Copy code
     ```
     db.collectionName.insertOne({ name: "John", age: 30 });
     ```
2. **Read**:
   - Use the find() method to retrieve documents from a collection. You can use a query to filter the documents.

     javascript
     Copy code
     ```
     db.collectionName.find({ name: "John" });
     ```
3. **Update**:
   - Use the updateOne(), updateMany(), or replaceOne() methods to update existing documents.

     javascript
     Copy code
     ```
     db.collectionName.updateOne({ name: "John" }, { $set: { age: 31 } });
     ```
4. **Delete**:
   - Use the deleteOne() or deleteMany() methods to remove documents from a collection.

     javascript
     Copy code
     ```
     db.collectionName.deleteOne({ name: "John" });
     ```

Create: Adding New Documents

To add documents to the users collection, we use the insertOne() method for a single document or insertMany() for multiple documents.

• **Operation:**

```
javascript
Copy code
db.users.insertOne({ name: "John", age: 30, city: "New York" });
db.users.insertOne({ name: "Jane", age: 25, city: "Los Angeles" });
db.users.insertMany([
  { name: "Alice", age: 28, city: "Chicago" },
  { name: "Bob", age: 35, city: "San Francisco" }
]);
```

- **Result:**

```
json
Copy code
[
  { "_id": ObjectId("..."), "name": "John", "age": 30, "city": "New York" },
  { "_id": ObjectId("..."), "name": "Jane", "age": 25, "city": "Los Angeles" },
  { "_id": ObjectId("..."), "name": "Alice", "age": 28, "city": "Chicago" },
  { "_id": ObjectId("..."), "name": "Bob", "age": 35, "city": "San Francisco" }
]
```

2. Read: Retrieving Documents

To retrieve documents from the users collection, we use the find() method. We can filter results with queries.

- **Operation:** Find all users named "John".

```
javascript
Copy code
db.users.find({ name: "John" });
```

- **Result:**

```
json
Copy code
[
  { "_id": ObjectId("..."), "name": "John", "age": 30, "city": "New York" }
]
```

- **Operation:** Find all users.

  javascript
  Copy code
  ```
  db.users.find();
  ```
- **Result:**

  json
  Copy code
  ```
  [
    { "_id": ObjectId("..."), "name": "John", "age": 30, "city": "New York" },
    { "_id": ObjectId("..."), "name": "Jane", "age": 25, "city": "Los Angeles" },
    { "_id": ObjectId("..."), "name": "Alice", "age": 28, "city": "Chicago" },
    { "_id": ObjectId("..."), "name": "Bob", "age": 35, "city": "San Francisco" }
  ]
  ```

3. Update: Modifying Existing Documents

To update existing documents, we use updateOne(), updateMany(), or replaceOne(). Updates can target specific fields using the $set operator.

- **Operation:** Update John's age to 31.

  javascript
  Copy code
  ```
  db.users.updateOne({ name: "John" }, { $set: { age: 31 } });
  ```
- **Result:**

  json
  Copy code
  ```
  [
    { "_id": ObjectId("..."), "name": "John", "age": 31, "city": "New York" },  //
  Updated
    { "_id": ObjectId("..."), "name": "Jane", "age": 25, "city": "Los Angeles" },
    { "_id": ObjectId("..."), "name": "Alice", "age": 28, "city": "Chicago" },
    { "_id": ObjectId("..."), "name": "Bob", "age": 35, "city": "San Francisco" }
  ]
  ```

- **Operation:** Update all users in "Los Angeles" to "San Diego".

  javascript
  Copy code
  ```
  db.users.updateMany({ city: "Los Angeles" }, { $set: { city: "San Diego" } });
  ```
- **Result:**

  json
  Copy code
  ```
  [
    { "_id": ObjectId("..."), "name": "John", "age": 31, "city": "New York" },
    { "_id": ObjectId("..."), "name": "Jane", "age": 25, "city": "San Diego" }, //
  Updated
    { "_id": ObjectId("..."), "name": "Alice", "age": 28, "city": "Chicago" },
    { "_id": ObjectId("..."), "name": "Bob", "age": 35, "city": "San Francisco" }
  ]
  ```

4. Delete: Removing Documents

To delete documents, we use deleteOne() or deleteMany().

- **Operation:** Delete user "Bob".

  javascript
  Copy code
  ```
  db.users.deleteOne({ name: "Bob" });
  ```
- **Result:**

  json
  Copy code
  ```
  [
    { "_id": ObjectId("..."), "name": "John", "age": 31, "city": "New York" },
    { "_id": ObjectId("..."), "name": "Jane", "age": 25, "city": "San Diego" },
    { "_id": ObjectId("..."), "name": "Alice", "age": 28, "city": "Chicago" }
  ]
  ```
- **Operation:** Delete all users older than 30.

javascript
Copy code
```
db.users.deleteMany({ age: { $gt: 30 } });
```
- **Result:**

json
Copy code
```
[
  { "_id": ObjectId("..."), "name": "Jane", "age": 25, "city": "San Diego" },
  { "_id": ObjectId("..."), "name": "Alice", "age": 28, "city": "Chicago" }
]
```

What is NOSQL Graph database? Explain Neo4j.

NoSQL Graph Database

Graph Databases are a type of NoSQL database that store data in a graph format. In graph databases:

- **Nodes** represent entities (like people, products, or places).
- **Edges** represent relationships between these entities (like "friends with" or "purchased").
- **Properties** are key-value pairs that provide additional information about nodes and edges (like a person's name or age).

Graph databases are great for applications that involve complex, interconnected data, such as social networks, recommendation systems, and fraud detection.

Neo4j

Neo4j is a widely-used graph database that uses a **property graph model** to represent data. It stores data as nodes and edges (relationships), with properties associated with both. Neo4j uses **Cypher**, a powerful query language designed specifically for working with graphs, which makes it easy to create, read, update, and delete data.

Key Features of Neo4j

1. **Flexible Schema**: Nodes and relationships can have any number of properties and don’t require a fixed schema.
2. **High Performance**: Designed for fast querying of graph data, enabling efficient traversals.
3. **ACID Compliance**: Provides strong consistency and reliable transactions.
4. **Scalability**: Supports scaling both vertically (improving server capacity) and horizontally (adding more servers).

Basic Cypher Query Examples in Neo4j

Here are some basic examples to help you understand how to use Neo4j and its Cypher query language:

1. **Create a Node**: Add a new person named "Alice" with age 30.

   cypher
   Copy code
   ```
   CREATE (n:Person {name: 'Alice', age: 30});
   ```
   ○ **Result**: A new node Person with properties name and age is created.
2. **Create a Relationship**: Create a "FRIEND" relationship between "Alice" and "Bob".

   cypher
   Copy code
   ```
   MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
   CREATE (a)-[:FRIEND]->(b);
   ```
   ○ **Result**: A relationship "FRIEND" is created from "Alice" to "Bob".
3. **Read Data**: Retrieve all people’s names and ages.

   cypher
   Copy code
   ```
   MATCH (n:Person) RETURN n.name, n.age;
   ```

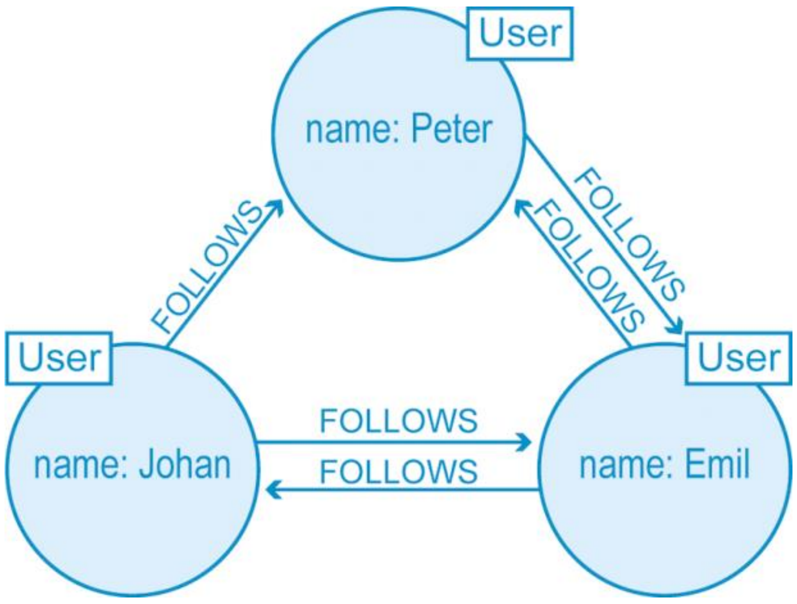
- ○ **Result**: Returns a list of all Person nodes with their name and age.

4. **Update a Node**: Change Alice's age to 31.

   cypher
   Copy code
   ```
   MATCH (n:Person {name: 'Alice'}) SET n.age = 31;
   ```
   - ○ **Result**: The age property of the node "Alice" is updated to 31.

5. **Delete a Node**: Remove the node for "Alice".

   cypher
   Copy code
   ```
   MATCH (n:Person {name: 'Alice'}) DELETE n;
   ```
   - ○ **Result**: The node for "Alice" is deleted from the graph.


User
name: Peter
User
name: Johan
User
name: Emil
FOLLOWS
FOLLOWS
FOLLOWS
FOLLOWS
FOLLOWS