Database

adcw263

is foreign key

ตัวเอียง foreign key
_ is primary key
สี่เหลี่ยมคือ table
เพชร คือ relatioin

These terms refer to different approaches to replicating data in distributed systems.

Eager, primary copy: In this approach, all updates to the data are made at a single primary copy, and the changes are immediately propagated to all other replicas. This ensures that all replicas have the same data at all times, but it can also result in higher network traffic and latency.

Eager, update anywhere: Here, updates can be made at any replica, and the changes are propagated immediately to all other replicas. This approach can reduce network traffic and latency, but it requires more coordination to ensure that all replicas are in sync.

Lazy, primary copy: In this approach, updates are made at a single primary copy, but changes are only propagated to other replicas when necessary (i.e., when a replica requests an update or when the primary copy detects a failure). This approach can reduce network traffic and latency, but it can also result in replicas being out of sync for some period of time.

Lazy, update anywhere: This approach allows updates to be made at any replica, but changes are only propagated to other replicas when necessary. This approach can reduce network traffic and latency, but it requires more coordination to ensure that all replicas are eventually updated.

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. It is a set of properties that ensure database transactions are processed reliably.

Atomicity: Atomicity ensures that all operations within a transaction are treated as a single unit of work, which either succeeds completely or fails completely. If a transaction fails, any changes made to the database must be rolled back to the state it was in before the transaction started.

Consistency: Consistency ensures that a transaction brings the database from one valid state to another. It ensures that data written to the database follows predefined rules, and that the data remains in a valid state after each transaction.

Isolation: Isolation ensures that concurrent transactions do not interfere with each other. Each transaction must be isolated from other transactions so that they can operate on the same data without causing conflicts or inconsistencies.

Durability: Durability ensures that once a transaction is committed, it will persist even in the event of a system failure or power outage. The changes made to the database must be stored and remain available, even if the system crashes or restarts.

A lock manager is a component of a database management system (DBMS) that controls access to shared resources in a multi-user environment. In a database system, multiple transactions may attempt to access the same data simultaneously, and without proper coordination, this can lead to inconsistencies or data corruption. To prevent this, the lock manager enforces a set of rules to ensure that only one transaction at a time can modify a particular data item.

The lock manager maintains a list of all locks held by each transaction, as well as a list of all pending lock requests. When a transaction requests a lock on a data item, the lock manager checks whether the requested lock conflicts with any existing locks held by other transactions. If there is no conflict, the lock is granted, and the transaction can proceed. If there is a conflict, the lock manager may block the requesting transaction until the conflicting lock is released.

There are different types of locks that can be used by the lock manager, including shared locks and exclusive locks. **Shared locks** allow **multiple transactions to read the same data** item simultaneously, while **exclusive locks prevent any other transactions from accessing the data item until the lock is released.** The lock

manager must carefully balance the need for concurrency with the need for consistency, to ensure that transactions can complete successfully while maintaining the integrity of the database.

Concurrent control in a database refers to the techniques and mechanisms used to manage multiple transactions accessing the database concurrently. In a multi-user environment, it is common for several transactions to access the same data at the same time. Without proper coordination, this can lead to data inconsistencies, incorrect results, or even data corruption.

Concurrent control ensures that transactions **execute in a consistent and isolated manner, without interfering with each other.** The goal of concurrent control is to **allow multiple transactions to access the database simultaneously** while ensuring that the results are consistent with the expectations of each transaction. There are two main approaches to concurrent control: locking-based and timestamp-based.

Locking-based concurrency control involves the **use of locks to coordinate access to data**. When a transaction accesses a data item, it requests a lock on that item, and the lock manager grants the lock if there are no conflicts with other transactions. If a transaction cannot obtain a lock, it may have to wait until the lock is released by another transaction. This approach ensures that no two transactions can modify the same data item simultaneously, preventing data inconsistencies.

Timestamp-based concurrency control uses **timestamps to order transactions and resolve conflicts.** Each transaction is assigned a unique timestamp when it starts, and the database keeps track of the order in which transactions execute. When a transaction accesses a data item, the database checks its timestamp to ensure that it has the most recent version of the data. If two transactions attempt to modify the same data item, the one with the earlier timestamp is rolled back, ensuring that the transactions execute in a serializable order.

Both locking-based and timestamp-based concurrency control are widely used in modern database management systems to ensure that transactions execute correctly and efficiently in a multi-user environment.

In database systems, there are four isolation levels that determine how transactions are allowed to read and modify data. These isolation levels are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE.

1. **READ UNCOMMITTED**: In this isolation level, transactions can read data that has been modified by other transactions but not yet committed. This means that transactions may read dirty data, which can result in inconsistent or incorrect results.
2. **READ COMMITTED**: In this isolation level, transactions can only read data that has been committed by other transactions. This ensures that transactions do not read dirty data, but it allows for non-repeatable reads, where a transaction may see different results when reading the same data multiple times.
3. **REPEATABLE READ**: In this isolation level, a transaction guarantees that it will see a consistent snapshot of the database throughout its execution. This means that a transaction will not see any changes made by other transactions after it has started, and other transactions will not see changes made by this transaction until it is committed. However, phantom reads may occur, where a transaction sees additional rows that were not visible in its initial snapshot.
4. **SERIALIZABLE**: In this isolation level, transactions are executed as if they were executed serially, one after the other. This ensures that transactions do not see any changes made by other transactions until they are committed, and it prevents phantom reads. However, this isolation level may result in a higher level of contention, where transactions may need to wait for locks to be released, which can impact performance.

Choosing the appropriate isolation level depends on the requirements of the application and the database system's capabilities. A higher isolation level provides more consistency and accuracy, but it can impact performance and concurrency, while a lower isolation level may provide better performance but can result in inconsistencies and incorrect results.

Database replication is the process of copying data from one database instance to another, often to provide increased availability, fault tolerance, or scalability. There are two common approaches to database replication: kernel-based replication and middleware-based replication.

Kernel-based replication involves replicating data at the operating system level, using features such as disk mirroring or disk-level replication. This approach is typically **performed by the operating system or hardware vendors** and can provide high availability and fault tolerance. However, it is **limited in terms of flexibility** and may not provide all of the features required for advanced replication scenarios.

Middleware-based replication, on the other hand, **uses a separate software layer or middleware to replicate data between database instances.** This approach provides more flexibility and can support more advanced replication scenarios, such as multi-master replication or conflict resolution. Middleware-based replication can also be used to replicate data between different types of databases, such as Oracle to MySQL or vice versa.

In both kernel-based and middleware-based replication, there are primary and replica database instances. The primary database instance is the master copy of the data, and any changes made to the data are propagated to the replica instances. The replica instances are read-only copies of the data, which can be used for reporting, backups, or failover in case of primary database failure.

Overall, the choice between kernel-based and middleware-based replication depends on the specific requirements of the database environment, including performance, scalability, and availability needs, as well as the desired level of flexibility and functionality.

The Two-Phase Commit (2PC) protocol is **a distributed transaction protocol** used to ensure atomicity and consistency of transactions across multiple participating nodes or databases in a distributed environment.

In a distributed environment, a transaction may involve multiple databases or nodes that need to coordinate to ensure that the transaction is either committed or rolled back consistently across all participating nodes. The 2PC protocol ensures that all participating nodes reach a consensus to either commit or abort the transaction.

The 2PC protocol is composed of two phases:

1. **The Prepare Phase**: In this phase, **the transaction coordinator sends a prepare request to all participating nodes.** Each participating node then checks whether it can commit the transaction or not. If a participating node cannot commit the transaction, it sends an abort message to the transaction coordinator. If all nodes can commit the transaction, they reply with a "yes" vote to the coordinator.
2. **The Commit Phase: If all participating nodes reply with a "yes" vote during the prepare phase, the coordinator sends a commit message to all nodes.** Upon receipt of the commit message, each participating node commits the transaction and releases any resources held during the transaction. If any node fails to receive the commit message or fails to commit the transaction, it sends an abort message to the coordinator.

If a node fails during the 2PC protocol, the protocol can be restarted to ensure that the transaction is either committed or rolled back consistently across all participating nodes. The 2PC protocol ensures that all participating nodes agree to commit or roll back a transaction, providing a consistent and reliable way to manage distributed transactions.