**Transactions:**

1. **Isolation levels:** Transactions running concurrently can run into race conditions:
   1. **Dirty Writes:** T1 overwrites the value written by T2 which has not been committed yet
   2. **Dirty Reads:** T1 reads uncommitted value from a write of T2
   3. **Non Repeatable Reads:** T1 reads value v1 at certain time but a value v2 at a different time
   4. **Phantoms:** A phantom is a row that matches the search criteria but is not initially seen. For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 reexecutes the statement that reads the rows, it gets a different set of rows.

To prevent against race conditions, different isolation levels exist. Transaction isolation levels are a measure of the extent to which transaction isolation succeeds.

1. **Read Uncommitted**: Prevents Dirty writes. Hold Write locks for the entire duration of transaction, so that the other transaction cannot overwrite.
2. **Read Committed**: The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data. Transaction holds write locks for the entire duration of the transaction, while read locks on the current row. The transaction releases read locks when it moves off the current row. This results in non repeatable read. Because once the read lock is released for the row, other transaction T2 can update the row. If T2 reads the row again, it will get a different value.
3. **Repeatable Read**: Same as above but read lock is also held for the entire duration of the transaction. Because other transactions cannot update or delete these rows, the current transaction avoids any nonrepeatable reads. The transaction releases its locks when it is committed or rolled bac
4. **Serializable**: The transaction holds a read lock (if it only reads rows) or write lock (if it can update or delete rows) on the range of rows it affects Because other transactions cannot update or delete the rows in the range, the current transaction avoids any nonrepeatable reads. Because other transactions cannot insert any rows in the range, the current transaction avoids any phantoms. The transaction releases its lock when it is committed or rolled back.
5. **Implementing Isolation Levels, Locks:**
   1. **Pessimistic Locking:** Two phase locking, read locks and write locks. Read locks can be shared. But write locks can be held by only one transaction and blocks and read or writes to the data item.
   2. **Optimistic Locking:** Does not block and check for conflicts only at the end of a transaction. If conflict is detected, transaction is aborted. *multi-version concurrency control*7 (MVCC). With MVCC, the data store keeps multiple versions of a data item. Read- only transactions aren’t blocked by other transactions, as they can keep reading the version of the data that was committed at the time the transaction started. But, a transaction that writes to the store is aborted or restarted when a conflict is detected. For read heavy, use optimistic locks because no locks are acquired and application is not blocked. For write heavy, use pessimistic locking to prevent retrying the same transactions.
6. **Atomicity in the context of distributed transactions:** Transactions running on multiple nodes as one. For example bank deposit. The accounts may be in different banks so separate processes. So T1 to debit for Person1 and T2 to credit for Person2. T1 and T2 could be running in different nodes i.e. different data stores. Another example is a transaction that affects certain rows on node1 and certain rows on node2 in a sharded database. How do you ensure atomicity here? Easier to do on one machine because of the usage of locks. But not on different nodes

**Two Phase Commit:** Coordinator – Client application that initiated the transaction. Doesn’t have data. Other nodes are called as participants. Coordinator asks node to ‘prepare’. Data is flushed to the disk along with the state of the transaction. If the DB crashes and it comes back up, it can look up the state of the transaction and acquires locks etc needed to commit the transaction. First phase -> talk to all nodes and prepare. If all participants reply that they are ready to commit, the coordinator sends out a commit message to all participants ordering them to do so. In contrast, if any process replies that it’s unable to commit, or doesn’t respond promptly, the coordinator sends an abort request to all participants. Limitations: Two points of no return. If the coordinator dies, the participants are stuck with the locks etc. The participant can’t make progress until it receives the commit or abort message from the coordinator later when it recovers. Second, once the coordinator makes up it’s mind, it has to see through that the transaction either succeeds or aborts. If a participant has crashed temporarily, it has to keep retrying until the request succeeds. So If either the coordinator or a participant fails, then all processes part of the transactions are blocked until the failing process comes back online. To make it more resilient, the coordinator and participants could be replicated.

**Asynchronous Transactions:** 2PC is not useful if the transactions are long because of the blocking nature. 2PC is useful for short lived transactions. What If the individual transactions could take a long time. For example, booking a connecting flight. You cannot hold up resources or locks for that long. Isolation guarantee needs to be relaxed. While 2PC use 2PL to provide isolation, Sagas would be using optimistic locking mechanisms. They provide eventually consistency. Example of Sagas, a transaction composed of booking a flight followed by booking a hotel. We have T1, T2, …., Tn and corresponding compensating transactions used in case of rollback. Implemented with the help of orchestrator. The coordinator can communicate asynchronously with the par- ticipants via message channels to tolerate temporarily unavailable ones.

Diagram

Description automatically generated