Weak Isolation Levels

Background

Isolation/Serializability is often too great of a performance penalty for databases to implement - instead many have chosen to offer weaker guarantees as to how they handle certain concurrency bugs.

Let’s go through some concurrency bugs, and look at strategies by databases to avoid them without providing complete transaction isolation.

Types of Concurrency Bugs

- Dirty Reads
- Dirty Writes
- Read skew (Non-repeatable reads)
- Lost Updates
- Write Skew and Phantoms

Dirty Reads

Clients should not be able to read data that has been written but not yet been committed, if they could they may see the database in an inconsistent state or read some data right before it gets rolled back

Dirty Reads

Clients should not be able to read data that has been written but not yet been committed, if they could they may see the database in an inconsistent state or read some data right before it gets rolled back

Solution: Database remembers old value of a write until said write is committed, do not use locks as one long transaction could make many reads wait a long time until it completes

kim’s_man: Kanye

Dirty Reads

Clients should not be able to read data that has been written but not yet been committed, if they could they may see the database in an inconsistent state or read some data right before it gets rolled back

Solution: Database remembers old value of a write until said write is committed, do not use locks as one long transaction could make many reads wait a long time until it completes

kim’s_man: Kanye -> Pete

Dirty Reads

Clients should not be able to read data that has been written but not yet been committed, if they could they may see the database in an inconsistent state or read some data right before it gets rolled back

Solution: Database remembers old value of a write until said write is committed, do not use locks as one long transaction could make many reads wait a long time until it completes

kim’s_man: Pete

Dirty Writes

When writing data, we can only overwrite committed data.

Dirty Writes

When writing data, we can only overwrite committed data.

Solution: Have a lock on each object, any transaction that wants to write an object must first grab the lock.

Read Committed Isolation

The weakest of the isolation levels that we will speak about, read committed isolation simply prevents dirty reads and dirty writes, allowing the other concurrency bugs to occur.

Read Skew

A client can make multiple reads to the database, however throughout the course of those reads, the database can change. As a result, the client sees the database in an inconsistent state.

This is problematic for things like analytics queries, as the data will make no sense if some pieces of it are more or less updated than others.

Read Skew

A client can make multiple reads to the database, however throughout the course of those reads, the database can change. As a result, the client sees the database in an inconsistent state.

Imagine reading bank account balances: originally I have $100 and so does my friend. After I read my balance, my friend transfers me $50, and then I read his balance. So now it looks like I still have $100 and my friend has $50, where did the other $50 go?

Solution: Snapshot Isolation

Snapshot Isolation

Every transaction is assigned an increasing transaction ID, and when writing a value, the transaction ID that wrote it is saved with it. When a transaction performs a read, it takes the value with the highest transaction ID that is less than the reader ID.

Jordan: [(cute, 1), (handsome, 8), (brollic, 14)]

Bieber: [(gross, 2), (uglier_than_jordan, 12), (ok_fine_good_looking, 19)]

Transaction 15 reads the values for Jordan and Bieber:

Snapshot Isolation

Every transaction is assigned an increasing transaction ID, and when writing a value, the transaction ID that wrote it is saved with it. When a transaction performs a read, it takes the value with the highest transaction ID that is less than the reader ID.

Jordan: [(cute, 1), (handsome, 8), (brollic, 14)]

Bieber: [(gross, 2), (uglier_than_jordan, 12), (ok_fine_good_looking, 16)]

Transaction 15 reads the values for Jordan and Bieber: Jordan = brollic, Bieber = uglier_than_jordan

Snapshot Isolation

Every transaction is assigned an increasing transaction ID, and when writing a value, the transaction ID that wrote it is saved with it. When a transaction performs a read, it takes the value with the highest transaction ID that is less than the reader ID.

Jordan: [(cute, 1), (handsome, 8), (brollic, 14)]

Bieber: [(gross, 2), (uglier_than_jordan, 12), (ok_fine_good_looking, 16)]

Transaction 15 reads the values for Jordan and Bieber: Jordan = brollic, Bieber = uglier_than_jordan

Without Snapshot Isolation, if T16 committed before T15 reads Bieber, T15 would see that Bieber = ok_fine_good_looking

Lost Updates

When two threads read an object, perform an operation on it, and then write it back, one of the threads' update may be lost

Jordan_net_worth: 1000000

Lost Updates

When two threads read an object, perform an operation on it, and then write it back, one of the threads' update may be lost

Jordan_net_worth: 1000000

T1: read jordan_net_worth and add 1000000

T2: read_jordan_net_worth and add 1000000

Lost Updates

When two threads read an object, perform an operation on it, and then write it back, one of the threads' update may be lost

Jordan_net_worth: 1000000

T1: read jordan_net_worth (1000000) and add 1000000 = 2000000

T2: read_jordan_net_worth (1000000) and add 1000000 = 2000000

Lost Updates

When two threads read an object, perform an operation on it, and then write it back, one of the threads' update may be lost

Jordan_net_worth: 2000000

I've been conned out of a million dollars

Solution: Atomic write operations, explicit locking, or automatic database detection

Solutions to Lost Updates

- Atomic write operations
  - Atomic counters, compare_and_set(old_val, new_val) - use exclusive locks
- Explicit locks in application code (“FOR UPDATE”) in SQL
  - Hard to reason about, leads to many bugs, avoid when possible
- Automatic database detection
  - Use snapshot isolation to automatically detect lost updates (can see the value that it was about to write has since changed), and rollback and retry
  - Good because it reduces the chance of writing buggy locking code

These techniques do not work in multileader/leaderless replication, as they assume one copy of the data, instead better to store conflicts as siblings and use custom resolution logic.

Write Skew

Two transactions each read the same set of objects in order to make a decision on whether to make a write, and then change different members of the set of objects which breaks some invariant.

id	Name	Table
1	Jordan	1
2	Lindsay Lohan	1
3	Paris Hilton	None
4	Kate Upton	None
5	Megan Fox	None

Write Skew

Two transactions each read the same set of objects in order to make a decision on whether to make a write, and then change different members of the set of objects which breaks some invariant.

id	Name	Table
1	Jordan	1
2	Lindsay Lohan	1
3	Paris Hilton	None
4	Kate Upton	None
5	Megan Fox	None

T_Fox: Sees 2 people at table 1

T_Upton: Sees 2 people at table 1

Write Skew

Two transactions each read the same set of objects in order to make a decision on whether to make a write, and then change different members of the set of objects which breaks some invariant.

id	Name	Table
1	Jordan	1
2	Lindsay Lohan	1
3	Paris Hilton	None
4	Kate Upton	1
5	Megan Fox	1

T_Fox: Sets table to 1

T_Upton: Sets table to 1

Invariant broken! Only 3 people to a table!

Write Skew Solution

Just apply a lock to all the rows that you are reading to form the predicate, so that only one transaction can read them at a time.

But what if those rows don’t all yet exist?

Phantoms

Same issue as write skew, but invariants are broken when both transactions create a new row. Nothing to put a lock on!

Solution: Materialize Conflicts

Materializing Conflicts

Imagine I want to book a meeting in a given room, at a given time. I would first query the database, and then add my meeting slot as a new row to the bookings table. But I can’t stop anybody else from creating the row, because I have nothing to put a lock on!

Materializing conflicts: create a blank row for the meeting slot in the bookings table prematurely (at the start of the week for all bookings that week for example), so that transactions can apply a lock to it, and we don’t have an overbooking.

Weak Isolation Summary

- Read Committed
  - Prevents dirty reads, dirty writes
- Snapshot Isolation
  - Prevents read skew (non-repeatable read), can easily be adapted to detect lost updates
- Predicate Locks and Materializing Conflicts
  - Can be used to prevent write skew (and phantoms)

Overall, using these methods results in a much faster database than one that uses true serializable isolation, as the locking is relatively minimal in comparison and concurrency can be maximized. However, if the application can take the performance hit, serializable isolation will reduce the need to reason about concurrency.