Practical Guide to SQL Transaction Isolation

August 1, 2017 **Newsletter ↳**

You may have seen isolation levels in the documentation for your database, felt mildly uneasy, and went on with life. Few day-to-day examples of using transactions really mention isolation. Most use the database's default isolation level and hope for the best. It's a fundamental topic to understand however and you'll feel more comfortable if you dedicate some time to study this guide.

I have assembled this information from academic papers, the PostgreSQL docs, and discussions with colleagues to answer not just *what* isolation levels are, but *when* to use them for maximum speed while preserving application correctness.

Basic Definitions

To properly understand SQL isolation levels, we ought first to consider transactions themselves. The idea of a transaction comes from contract law: legal transactions must be atomic (either all provisions apply or none do), consistent (abiding by legal protocols), and durable (after commitment the parties cannot go back on their word). These properties are the A, C and D in the popular "ACID" acronym for database management systems. The final letter, "I" for isolation, is what this article is all about.

In databases, as opposed to law, a transaction is a group of operations that transform the database from one consistent state to another. This means that if all database consistency constraints were satisfied prior to running a transaction, then they will remain satisfied afterward.

Could the database have pushed this idea further and enforced constraints at each and every SQL data modification statement? Not with the available SQL commands. They are not expressive enough to allow the user to preserve consistency at every step. For instance, the

classic task of transferring money from one bank account to another involves a temporarily inconsistent state after debiting one account but prior to crediting the other. For this reason transactions, and not statements, are treated as the units of consistency.

At this point we can imagine transactions running serially on the database, each waiting its turn for exclusive access to the data. In this orderly world the database would move from one consistent state to another, passing through brief periods of harmless inconsistency.

However the utopia of serialized transactions is infeasible for virtually any multi-user database systems. Imagine an airline database locking access for *everyone* while one customer books a flight.

Thankfully truly serialized transaction execution is usually unnecessary. Many transactions have nothing to do with one another because they update or read entirely separate information. The final result of running such transactions at the same time – of interleaving their commands – is indistinguishable from choosing to run one entire transaction before the other. In this case we call them *serializable.*

However running transactions concurrently does pose the danger of conflicts. Without database oversight the transactions can interfere with each other's working data and can observe incorrect database state. This can cause incorrect query results and constraint violation.

Modern databases offer ways to automatically and selectively delay or retry commands in a transaction to prevent interference. The database offers several modes of increasing rigor for this prevention, called isolation levels. The "higher" levels employ more effective – but more costly – measures to detect or resolve conflicts.

Running concurrent transactions at different isolation levels allows application designers to balance concurrency and throughput. Lower isolation levels increase transaction concurrency at the risk of transactions observing certain kinds of incorrect database state.

Choosing the right level requires understanding which concurrent interactions pose a threat to the queries required by an application. As we will see, sometimes an application can get

away with a lower than normal isolation level through manual actions like taking explicit locks.

Before examining isolation levels, let's take a stop at the zoo to see transaction problems in captivity. The literature calls these problems "transaction phenomena."

The Zoo of Transaction Phenomena

For each phenomenon we examine the telltale pattern of interleaved commands, see how it can be bad, and also note times when it can be tolerated or even used intentionally for desirable effects.

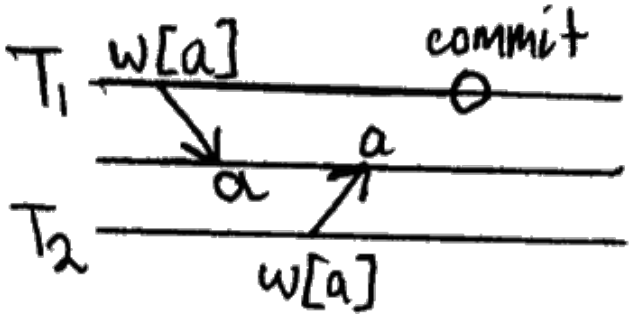
We'll use a shorthand notation for the actions of two transactions T1 and T2. Here are some examples:

- `r1[x]` – T1 reads value/row x
- `w2[y]` – T2 writes value/row y
- `c1` – T1 commits
- `a2` – T2 aborts

Dirty Writes

Transaction T1 modifies an item, T2 further modifies it before T1 commits or rolls back.

Pattern



Dangers

If dirty writes are permitted then it is not always possible for the database to roll back a transaction. Consider:

- {db in state A}
- w1[x]
- {db in state B}
- w2[x]

- {db in state C}
- a1

Should we go back to state A? No, because that would lose w2[x]. So we remain at state C. If c2 happens then we're good. However if a2 happens then what? We can't pick B or it would undo a1. But we can't pick C because that would undo a2. Reductio ad absurdum.

Because dirty writes break the atomicity of transactions, no relational database allows them at even the lowest isolation level. It's simply instructive to consider the problem in the abstract.

Dirty writes also allow a consistency violation. For instance suppose the constraint is x=y. The transactions T1 and T2 might individually preserve the constraint, but running together with a dirty write violate it:

- start, x = y = 0
- w1[x=1] … w2[x=2] … w2[y=2] … w1[y=1]
- now x = 2 ≠ 1 = y

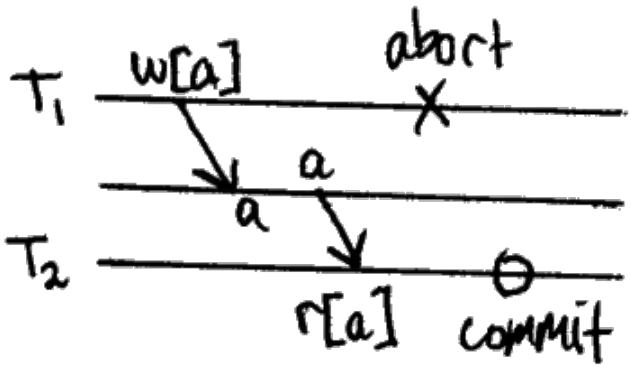
Legitimate Uses

There is no situation where dirty writes are useful, even as a shortcut. Hence no database allows them.

Dirty Reads

A transaction reads data written by a concurrent uncommitted transaction. (As in the previous phenomenon, uncommitted data is called "dirty.")

Pattern



Dangers

Say T1 modifies a row, T2 reads it, then T1 rolls back. Now T2 is holding a row that “never existed.” Basing future decisions off of nonexistent data can be a bad idea.

Dirty reads also open the door for a constraint violation. Assume the constraint x=y. Also suppose T1 adds 100 to both x and y, and T2 doubles them both. Either transaction alone preserves x=y. However a dirty read of w1[x += 100], w2[x *= 2], w2[y *= 2], w1[y += 100] violates the constraint.

Finally, even if no concurrent transactions roll back, a transaction starting in the middle of another's operation can dirty read an inconsistent database state. We would prefer that transactions could count on being started in a consistent state.

Legitimate Uses

Dirty reads are useful when one transaction would like to spy on another, for instance during debugging or progress monitoring. For instance, repeatedly running COUNT(*) on a table from one transaction while another ingests data into it can show the ingestion speed/progress, but only if dirty reads are allowed.

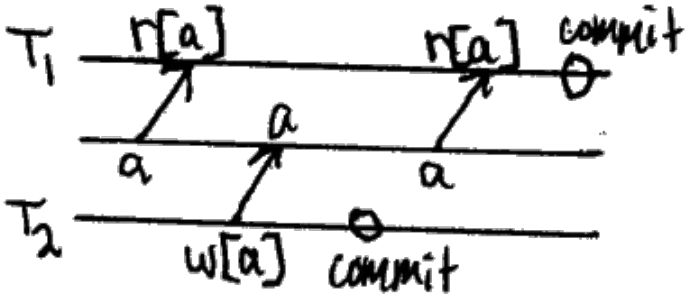
Also this phenomenon won't happen during queries for historical information that has long ceased changing. No writes, no problems.

Non-Repeatable Reads, and Read Skew

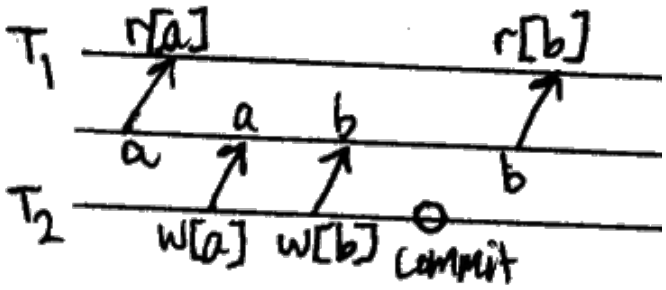
A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that has committed since the initial read).

Note that this differs from a dirty read in that the other transaction has committed. Also this phenomenon requires two reads to manifest.

Pattern



The form involving two values is called *read skew:*



Non-repeatable read is a degenerate case where b=a.

Dangers

Like dirty reads, non-repeatable reads allow a transaction to read an inconsistent state. It happens in a slightly different way. Suppose the constraint is x=y.

- start, x = y = 0
- r1[x] … w2[x=1] … w2[y=1] … c2 … r1[y]
- From T1's perspective, x = 0 ≠ 1 = y

T1 never read any dirty data, but T2 slipped in, changed values and committed between T1's reads. Notice this violation didn't even involve T1 re-reading the same value.

Read skew can cause constraint violations between two related elements. For instance, assume the constraint x+y > 0. Then:

- start, x = y = 50
- r1[x] … r1[y] … r2[x] … r2[y] … w1[y=-40] … w2[x=-40] … c1 … c2
- T1 and T2 each observe x+y=10, but together they make it -80.

Another constraint violation involving two values is that between a foreign key and its target. Read skew can mess that up too. For instance, T1 could read a row from table A pointing at table B. Then T2 can delete that row from B and commit. Now A believes the row exists in B but will be unable to read it.

This would be catastrophic when taking a database backup while other transactions are running since the observed state can be inconsistent and unsuitable for restoration.

Legitimate Uses

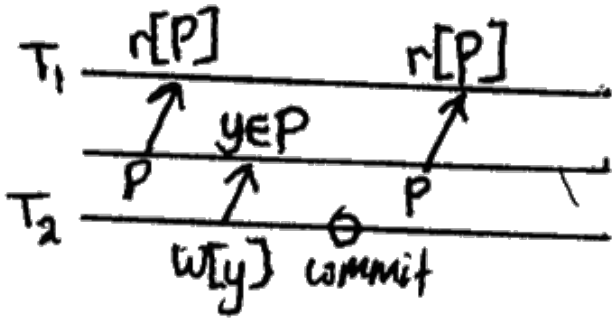
Doing non-repeatable reads allows access to the freshest committed data. This might be useful for large (or frequently repeated) aggregate reports when they can tolerate reading ephemeral constraint violations.

Phantom Reads

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

This is similar to a non-repeatable read except it involves a changing collection matching a predicate rather than a single item.

Pattern



Dangers

One situation is when a table contains rows that represent resource allocation (like employees and their salaries) where one transaction, “the adjuster,” increases the resources per row while another transaction inserts new rows. A phantom read will include the new rows, causing the adjuster to blow the budget.

Or a related example. Consider a constraint that says a set of job tasks determined by a predicate cannot exceed a sum of eight hours. T1 reads this predicate, determines the sum is only seven hours and adds a new task of one hour duration, while a concurrent transaction T2 does the same thing.

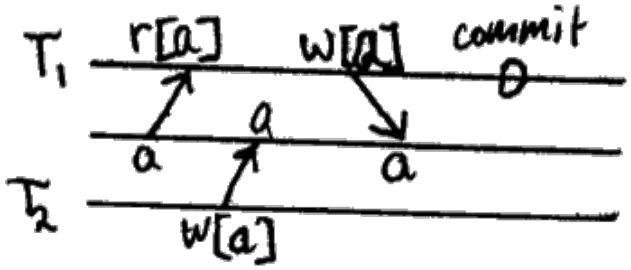
Legitimate Uses

Paginated search results may benefit from including brand new items as the page turns. Then again, inserted or deleted items can shift which items are on which pages as the user navigates.

Lost Update

T1 reads an item. T2 updates it. T1 updates it possibly based on what it read, and commits. T2’s update has been lost.

Pattern



Dangers

In some ways this almost doesn't feel like an anomaly. It can't violate database constraints because the end result is that some work simply hasn't been performed. Also it's similar to when an application blindly but serially updates the same value twice.

However it is an anomaly after all because there is zero chance for any other transaction to see the update, and T2's commit behaves like a rollback. In any serial execution *somebody* else would see the change, or at least could have if they checked.

Lost update has particularly bad effects when applications perform actions in the real world between the read and write.

For instance two people simultaneously try to buy the last available ticket for an event, spawning two transactions which read the there is one remaining ticket for sale. The app in separate threads queues emails with printable tickets, and updates the remaining ticket count to zero. After both updates happen, there are zero tickets remaining which is correct. However one of the customers has an email for a duplicate ticket.

Finally, note that the risk of lost updates increases when applications (usually through an ORM) update all columns in a row rather than just those columns that have changed since a read.

Legitimate Uses

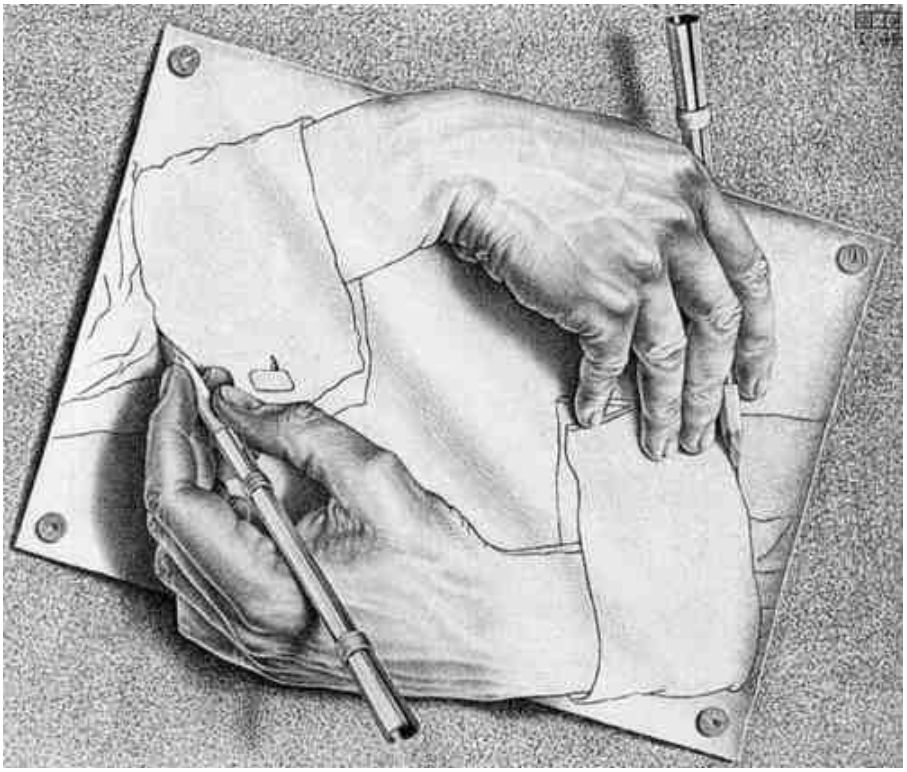
Lost updates can't happen with atomic read-update statements like `UPDATE foo SET bar`

`= bar + 1 WHERE id = 123;` because there is no way for another transaction to slip a write between the reading and incrementing of bar. The phenomenon happens when an application reads an item, performs internal calculations on it, and then writes a new value. But we'll get into that later.

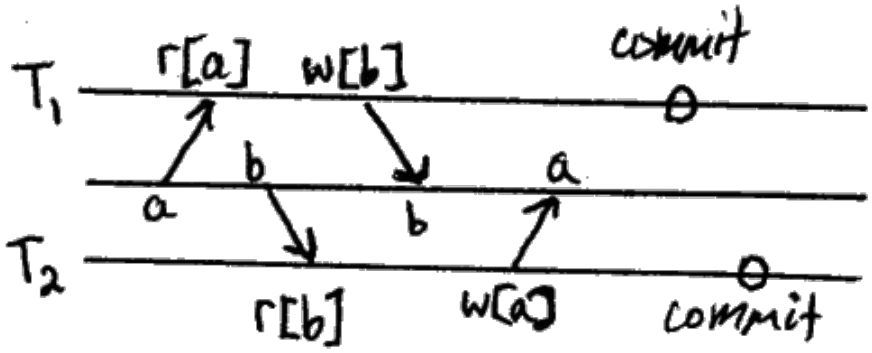
Sometimes an application may be fine with missing some values in a history of updates. Perhaps sensors are rapidly overwriting a measurement from multiple threads and we want to read only a reasonably recent value. This case, albeit slightly contrived, can tolerate lost updates.

Write Skew

Two concurrent transactions each determine what they are writing based on reading a data set which overlaps what the other is writing.



Pattern



Note that if b=a then we have a lost update.

Dangers

Write skew creates non-serializable transaction histories. Recall that this means no way of running the transactions one after another will produce the same result as their pathological interleaving.

The clearest example I've seen is the case of the black and white rows. Copying verbatim from a PostgreSQL wiki: In this case there are rows with a color column containing "black" or "white." Two users concurrently try to make all rows contain matching color values, but their attempts go in opposite directions. One is trying to update all white rows to black and the other is trying to update all black rows to white.

If these updates are run serially, all colors will match. However without any database protective measures the interleaved updates will simply reverse each other, leaving behind a mix of colors.

Write skew also breaks constraints. Suppose we constrain $x + y \geq 0$. Then

- start, x = y = 100
- r1[x] … r1[y] … r2[x] … r2[y] … w1[y=-y] … w2[x=-x]
- now x+y = -200

Both transactions read that x and y have value 100, so individually it's fine for each transaction to negate one the values, the total would still be non-negative. However negating both values results in x+y=-200, violating the constraint. For emotional gravity this is usually framed in terms of bank accounts where account balances are allowed to go negative as long as the sum of commonly held balances remains non-negative.

Read-Only Serialization Anomaly

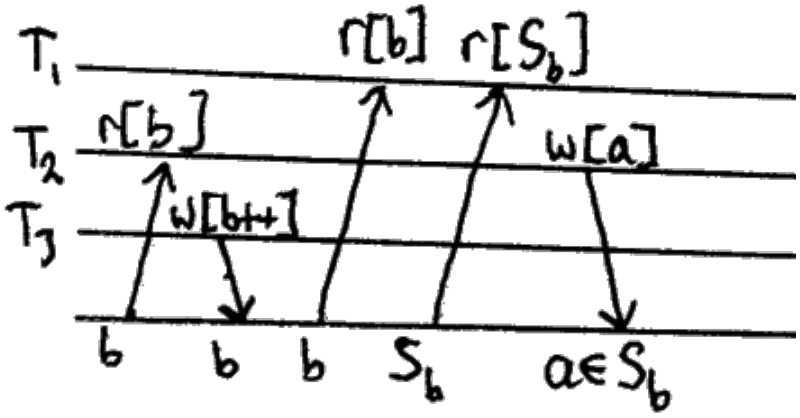
A transaction may see a control record updated to show that a batch has been completed but not see one of the detail records which is logically part of the batch because it read an earlier revision of the control record.

Whereas only two concurrent transaction sufficed to cause the previous anomalies, this one needs three. It attracted interest upon discovery in 2004 because it reveals a weakness in the snapshot isolation level (discussed later), as well as manifesting in the only transaction of the three that performs no writes.

Pattern

The transactions race to do three things,

- T1: generate report for current batch
- T2: add new receipt to current batch
- T3: make new batch the “current” one



Dangers

The history demonstrated above is not serializable. Running the transactions serially has the invariant that after a report transaction has shown the total for a particular batch, subsequent transactions cannot change that total.

Database consistency remains intact with this anomaly but the results of the report are incorrect.

Legitimate Uses

Given that it took until 2004 for anyone to notice the phenomenon, it’s not as likely to cause problems as prior phenomena in the zoo. There isn’t any time that it’s really desirable, but it may not be very severe either.

Others?

Have we identified all possible transaction phenomena? It can be tricky to tell; the ANSI SQL-92 standard thought they had covered everything with dirty read, non-repeatable read, and phantom read. It wasn't until 1995 that Berenson et al identified other serialization anomalies, and not until 2004 as noted that the read-only anomaly was documented.

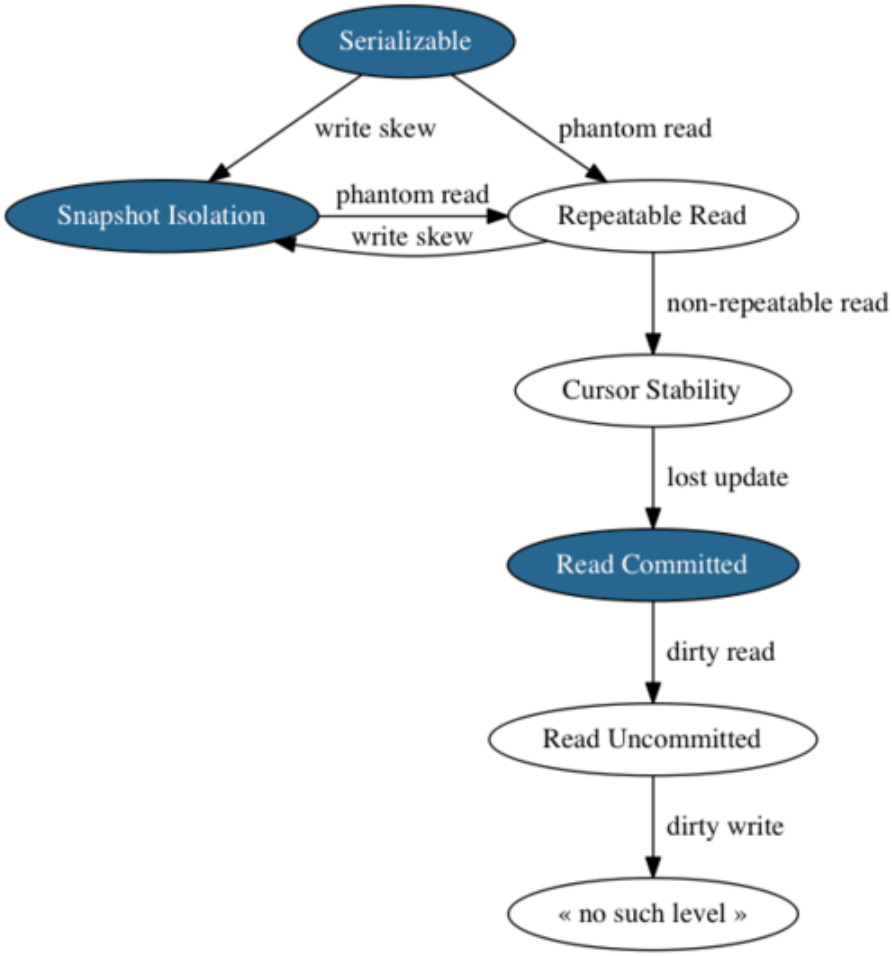
The first relational databases used locking to manage concurrency. The SQL standard talked in terms of transaction phenomena rather than locks to allow non-lock-based implementations of the standard. However the reason that the standard writers failed to uncover other anomalies was because the three they identified were "locks in disguise."

I personally don't know whether there are more undocumented transaction phenomena, but it seems doubtful. There are now numerous papers investigating the properties of serializability per se and it seems like the theoretical underpinnings are in place.

Isolation Levels

Commercial databases provide concurrency control with a range of isolation levels which are in fact controlled serializability violations. An application picks lower levels in order to achieve higher performance. Higher performance means better transaction execution rate and shorter average transaction response time.

If you have understood the "zoo" of concurrency problems in the previous section, then you're well on your way to knowing enough to intelligently choose the right isolation level for your application. Without getting too deep into *how* the levels prevent different phenomena, here is *what* each prevents.



At the top, Serializable doesn't allow any phenomena. Following the arrows removes protection for the labeled anomaly.

The three nodes in blue are the levels actually offered by PostgreSQL. What's confusing is that the SQL spec recognizes a limited number of levels, so PostgreSQL maps the names from the spec to the actual levels supported:

What you ask for	What you get
Serializable	Serializable
Repeatable Read	Snapshot Isolation
Read Committed	Read Committed
Read Uncommitted	Read Committed

For example:

```
BEGIN ISOLATION LEVEL REPEATABLE READ;

-- Now we're in snapshot isolation
```

Read committed is the default level, so imagine the concurrency problems your existing apps might be experiencing if you haven't taken precautions.

Optimistic vs Pessimistic

As mentioned, we're not going to dive into how each of PostgreSQL's isolation levels prevents concurrency phenomena, but we do need to understand there are two general approaches: optimistic and pessimistic concurrency control. It's important because each approach requires different application programming techniques.

Pessimistic concurrency control takes locks on database rows to force transactions to wait their turn reading and writing. It is "pessimistic" because it always takes the time to acquire and release locks, assuming gloomily that there will be contention.

Optimistic control doesn't bother taking locks, it just puts each transaction into a separate snapshot of the database state and watches for any contention to happen. If one transaction interferes with another, the database halts the offender and erases its work. This ends up being efficient when interference is rare.

The number of conflicts encountered depends on several factors:

- Contention for individual rows. The potential for conflicts increases as the number of transactions attempting to update the same row increases.
- Number of rows read in isolation levels which prevent non-repeatable reads. The more rows read, the greater the chance that some of these rows are updated by concurrent transactions.
- Size of the scan ranges used in isolation levels which prevent phantom reads. The larger the scan

ranges, the higher the chance that concurrent transactions will introduce phantom rows.

In PostgreSQL, two levels use optimistic concurrency control: the repeatable read (which is actually snapshot isolation) and serializable levels. **These levels are not magic fairy dust that you sprinkle on an unsafe app to fix its problems.** They require modifying application logic.

An application interacting with PostgreSQL using an isolation level with optimistic concurrency control must be built carefully. Remember that nothing is certain until commit, all work can be erased at a moment's notice. The app must be prepared to detect when its query has been halted with error 40001 (aka `serialization_failure`), and retry the transaction. Also an application should never perform an irreversible real-world action during such a transaction. The app must guard that kind of action with a pessimistic lock, or perform the action at the conclusion of a successful commit.

Also one might imagine catching the serialization exceptions and retrying them within a PL/pgSQL function, but the retry cannot happen there. The whole function runs *within* a transaction, and loses control of execution before commit is called. Unfortunately it is mostly at commit time when serialization errors happens, too late for the function to catch them.

Retries must be made by the database client. Many languages provide helper libraries for the task. Here are a few.

- Haskell: hasql-transaction (https://hackage.haskell.org/package/hasql-transaction) auto retries and runs in a monad which disallows unrepeatable side effects
- Python: psycopg2 how to retry (https://www.slideshare.net/petereisentraut/programming-with-python-and-postgresql)
- Ruby: auto-retrying in sequel (https://github.com/jeremyevans/sequel/blob/master/doc/transactions.rdoc#automatically-restarting-transactions) or the transaction_retry gem (https://github.com/qertoip/transaction_retry)

Because re-doing transactions can be wasteful, it's good to remember that simple

transactions with limited duration are most effective at avoiding lost work.

Compensating for Low Isolation Levels

Generally it's best to use an isolation level which protects against any anomalies that would disturb your queries. Let the database do what it does best. However if you believe that only certain anomalies will be happen in your situation, you can choose to use a lower isolation level with pessimistic locking.

For example we can prevent lost updates in a read committed transaction by taking a lock on the row between reading and updating it. Just add "FOR UPDATE" to the select statement.

```
BEGIN;

SELECT *
  FROM player
 WHERE id = 42
   FOR UPDATE;

-- some kind of game logic here

UPDATE player
   SET score = 853
 WHERE id = 42;

COMMIT;
```

Any other transaction which tries to also select that row for update will block until the first transaction is complete. This select for update trick is even useful in serializable transactions to avoid serialization errors which would require a retry, especially if you intend to perform non-idempotent application actions.

Finally, you can take a calculated risk in going for a lower level. The main reason for snapshot isolation's adoption is that it allows better performance than serializability, yet still avoids most of the concurrency anomalies that serializability avoids. If write skew is not

anticipated for your situation then you can turn the level down to snapshot.

Sources and Further Reading

Thanks to a number of people who have advised me in the writing of this article.

- On the #postgresql Freenode IRC channel: Andrew Gierth (RhodiumToad) and Vik Fearing (xocolatl)
- Personal conversations: Marco Slot, Andres Freund, Samay Sharma, and Daniel Farina from Citus Data (https://www.citusdata.com/)

Further reading

- Joe Celko's SQL for Smarties (http://shop.oreilly.com/product/9780128007617.do), chapter 2
- A Critique of ANSI SQL Isolation Levels (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf)
- Transaction Isolation (https://www.postgresql.org/docs/current/static/transaction-iso.html) in PostgreSQL docs
- A Read-Only Transaction Anomaly Under Snapshot Isolation (http://www.cs.umb.edu/~poneil/ROAnom.pdf)
- Serializable Snapshot Isolation in PostgreSQL (http://vldb.org/pvldb/vol5/p1850_danrkports_vldb2012.pdf)
- Data Consistency Checks at the Application Level (https://www.postgresql.org/docs/current/static/applevel-consistency.html) in the PostgreSQL docs
- The Transaction Concept: Virtues and Limitations (http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf)

Related Posts

- Deferrable SQL Constraints in Depth (2017-08-27-deferrable-sql-constraints.html)

- PostgreSQL Domain Integrity In Depth (2017-10-21-sql-domain-integrity.html)
- Faster PostgreSQL Counting (2016-10-12-count-performance.html)
- Relocatable PostgreSQL Builds (2016-05-21-relocatable-postgresql.html)
- Postgres Adores a Vacuum (2016-04-19-postgres-adores-vacuum.html)
- A Tour of PostgREST (2016-03-20-postgrest-tour.html)
- Database migrations without merge conflicts (2014-04-30-database-migrations-without-merge.html)