If Eventual Consistency Seems Hard, Wait Till You Try MVCC

Published Dec 8, 2014 by Baron Schwartz in Programming, Databases at https://www.xaprb.com/blog/2014/12/08/eventual-consistency-simpler-than-mvcc/

This should sound familiar:

> One of the great lies about NoSQL databases is that they're simple. Simplicity done wrong makes things a lot harder and more complicated to develop and operate. Programmers and operations staff end up reimplementing (badly) things the database should do.

Nobody argued this line of reasoning more vigorously than when trying to defend relational databases, especially during the darkest years (ca. 2009-2010), when NoSQL still meant **NO SQL DAMMIT**, all sorts of NoSQL databases were sprouting, and most of them were massively overhyped. But as valid as those arguments against NoSQL's "false economy" simplicity were and are, the arguments against relational databases' complexity hold true, too.

The truth is that no database is really simple. Databases have a lot of functionality and behaviors—even the "simple" databases do—and require deep knowledge to use well when reliability, correctness, and performance are important.

Eventual Consistency is Hard

Eventual consistency is hard to work with because developers bear extra burden. I suppose the Dynamo paper is the best source to cite:

> Dynamo targets the design space of an "always writeable" data store… This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected… The next design choice is who performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited…

One can trivially quote this out of context and argue that a bunch of database logic ends up being reimplemented in the application at read time, everywhere a read occurs. Indeed, sometimes this extreme does occur. Some use cases might actually need to check and reconcile conflicting updates with every single read.

You can find lots of other examples of this type of complexity in similar systems, such as the Riak documentation, which has lofty-sounding phrases like "causal context" and "dot-

ted version vectors." It does sound like one would need a PhD to use such a system, doesn't it?

When challenged in this way, many NoSQL advocates would respond that tradeoffs are necessary in distributed systems, and perhaps bring up the CAP Theorem, Jepsen and so forth. These kinds of topics are similar to Schroedinger's Cat, or double-slit experiments, or whatnot. Relatively ignorant people like me bring these up around the pool table and argue about them to try to sound smart, without knowing much about them.

Distributed systems are hard! There's no denying that. But is there a better way?

How Simple Are Relational Systems Anyway?

All this distributed systems theory and eventual consistency and so on… it's enough to make you long for the simplicity of a good old relational database, isn't it? "Everyone knows" that servers are massively powerful these days. Your favorite relational database of choice is claimed to be capable of scaling vertically to all but the most incredibly large-scale applications. So why not just do that, and keep it simple?

Let's talk about that word, simplicity.

Simplicity in relational systems is only achieved when there's no concurrency. Add in concurrency, and all the complexity of distributed systems comes home to roost, because distributed and concurrent are fundamentally about solving some of the same problems. In fact, unless you're running a single-writer, single-reader database on a single-core server—and maybe not even then, I'm not sure—you actually have a distributed system inside your server. Everything's distributed.

Preetam Jinka
@PreetamJinka

Sorry, I'm not impressed with serializable isolation via a single writer mutex.

Concurrent operation isn't a nice-to-have in most systems, it's a given. The way many relational systems handle concurrency is with this nifty little thing called Multi-Version Concurrency Control (MVCC). It's way simpler than eventual consistency. (Sarcasm alert!)

It works a little like this:

1. There are four standard transaction isolation levels, each with their own kinds of constraints and tradeoffs. Each defines which kinds of bad, inconsistent behaviors aren't allowed to happen.
2. In REPEATABLE READ, the isolation level that a lot of people consider ideal, you

get "read snapshots" that let you see an unchanging view of the database over time. Even as it's changing underneath you! This is implemented by keeping old row versions until they are no longer needed.
3. Other isolation levels, such as READ COMMITTED, are "bad." Because they don't protect you, the developer, from the complexity of the underlying implementation. And they don't allow you a true ACID experience.[1] A true ACID experience is about Atomicity, Consistency, Isolation, and Durability.
4. Back to REPEATABLE READ, the only isolation level that is approved by the Holy See. It's really simple. Everything appears just like you are the only user in the system. As a developer, you can just work with the database logically as you're supposed to, and you don't have to think about other transactions happening concurrently.

Clearly, this is much better than eventually consistent databases, right?

The Rabbit-Hole That Is MVCC

Unfortunately, the relational databases and their MVCCs are far from such a utopia. The reality is that MVCC is way more complex than I've described.

MVCC and the ACID properties are intertwined in very complex ways. The first problem comes from the ACID properties themselves. These four properties are almost universally misunderstood. It's almost as bad as the CAP theorem. I have to look up the definitions myself every single time. And then I always wind up asking myself, "what's the difference between Consistency and Isolation again?" Because the definitions seem like each one is halfway about the other, and there's no consistent way to think about them in isolation from each other.[2]

Next, isolation levels. Every database implements them differently. There's a lot of disagreement about the right way to implement each of the isolation levels, and this must have been an issue when the standards were written, because the standards leave a lot unspecified. Most databases are pretty opinionated, by contrast. Here's what PostgreSQL says (emphasis mine):

> The reason that PostgreSQL only provides three isolation levels is that this is *the only sensible way*[3] to map the standard isolation levels to the multiversion concurrency control architecture.

And MySQL, via InnoDB:

> InnoDB supports each of the transaction isolation levels described here using different locking strategies. You can enforce a high degree of consistency with the default REPEATABLE READ level, for operations on crucial data where ACID compliance is important. Or you can relax the consistency rules with READ COMMITTED or even READ UNCOMMITTED, in situations such as bulk reporting where precise

> consistency and repeatable results are less important than minimizing the amount of overhead for locking. SERIALIZABLE enforces even stricter rules than REPEATABLE READ, and is used mainly in specialized situations, such as with XA transactions and for troubleshooting issues with concurrency and deadlocks.

At a glance, it sounds like MySQL/InnoDB asserts that all four levels can be sensibly implemented, in contradiction to PostgreSQL's documentation. We'll dig into this more later. For the moment it's enough to note that InnoDB's MVCC behavior is more similar to Oracle's than it is to PostgreSQL's, but still, the docs say things like "A somewhat Oracle-like isolation level with respect to consistent (nonlocking) reads."

From experience I know that Microsoft SQL Server's locking and multiversion concurrency model is different yet again. So there's at least four different implementations with very different behaviors—and we haven't even gotten to other databases. For example, Jim Starkey's failed Falcon storage engine for MySQL was going to use "pure MVCC" in contradistinction to InnoDB's "mixed MVCC," whatever that means. Falcon, naturally, also had "quirks" in its MVCC implementation.

Serializable isolation is fairly clear, but understanding what the other systems actually provide is really hard. And even when you understand what they're supposed to provide, documentation and implementation bugs make it even worse.

Let's dig into a few of these implementations a bit and see what's really the situation.

InnoDB's MVCC

InnoDB's MVCC works, at a high level, by keeping old row versions as long as they're needed to be able to recreate a consistent snapshot of the past as the transaction originally saw it, and locking any rows that are modified.

There are at least four different scenarios to explore (one for each isolation level), and more in various edge cases. Quirks, let's call them.

The most obvious case we should look at is REPEATABLE READ, the default. It's designed to let you select a set of rows and then repeatedly see the same rows on every subsequent select, as long as you keep your transaction open. As the docs say,

> All consistent reads within the same transaction read the snapshot established by the first read.

Sounds elegant and beautiful. But it turns ugly really, really fast.

> For locking reads (SELECT with FOR UPDATE or LOCK IN SHARE MODE), UPDATE, and DELETE statements, locking depends on whether the statement uses a unique index with a unique search condition, or a range-type search condition. For a unique index with a unique search condition, InnoDB locks only the index record

> found, not the gap before it. For other search conditions, InnoDB locks the index range scanned, using gap locks or next-key locks to block insertions by other sessions into the gaps covered by the range.

What the hell just happened?

The abstraction just leaked, that's what.

The problem is due to several logical necessities and implementation details. It's not solely one or the other. The MVCC model is trying to balance a bunch of things going on concurrently, and there are logical contradictions that can't go away, no matter how sophisticated the implementation. There are going to be edge cases that have to be handled with special exceptions in the behavior. And the implementation details leak through, inevitably. That's what you are seeing above.

One of the logical necessities, for example, is that you can only modify the latest version of a row (eventually, at least). If you try to update an old version (the version contained in your consistent snapshot), you're going to get into trouble. There can (eventually) be only one truth, and conflicting versions of the data aren't allowed to be presented to a user as they are in eventual consistency. For this reason, various kinds of operations cause you to confront hard questions, such as:

1. Should the implementation disallow updating rows for which the snapshot has an out-of-date version, i.e. its version of reality has diverged from the latest version?
2. What is the latest version? Is it the latest committed version, the latest uncommitted version? What does "latest" mean? Is it "most recently updated by clock time" or is it "update by the transaction with the highest sequence number?" Does this vary between isolation levels?
3. If the implementation allows updating rows that are out-of-date (supposing the previous points have been resolved), what happens? Do you "leak" out of your isolation level, hence breaking consistency within your transaction? Do you fail the transaction? Or do you allow updating an old version, but then fail at commit time?
4. What happens if a transaction fails, and how does it fail / how is this presented to the user? (InnoDB used to deadlock and roll back the whole transaction; later it was changed to roll back just the failed statement).

Fundamentally you are going to run into problems such as these. And they have to be resolved, with various levels of confusion and complexity.

I should also note that InnoDB actually tries to go above and beyond the SQL standard. The standard allows phantom reads in REPEATABLE READ, but InnoDB uses next-key locking and gap locking to avoid this and bring REPEATABLE READ closer to SERIALIZABLE without the obnoxious locking implied by SERIALIZABLE. PostgreSQL does the same thing.

I've barely scratched the surface of the complexities of how InnoDB handles transactions, locking, isolation levels, and MVCC. I am not kidding. There is a large amount of documentation about it in the official manual, much of which requires serious study to understand. And beyond that, there is a lot that's not officially documented. For example, here's a blog post from one of the InnoDB authors that explains how various performance optimizations impact index operations. This might seem unrelated, but every access InnoDB makes to data has to interact with the MVCC rules it implements. And this all has implications for locking, deadlocks, and so on. Locking in itself is a complex topic in InnoDB. The list goes on.

How It Works In PostgreSQL

Sensibly, apparently ;-) Well, seriously, I have a lot less experience with PostgreSQL. But from the above it's quite clear that the PostgreSQL documentation writers could find lots of support for a claim that attempting to implement all four standard isolation levels, at least in the way that InnoDB does, is not sensible.

The PostgreSQL documentation, unlike the MySQL documentation, is largely limited to a single page.

Mark Callaghan
@markcallaghan

Read cursor isolation docs for Oracle, PG, InnoDB. PG docs are clear, others probably not. Tech writing is hard.

First of all, PostgreSQL uses READ COMMITTED by default. This means that if you SELECT some rows within a transaction, then wait while another transaction modifies them and commits, then SELECT them again, you'll see the changes. Whether this is OK is for you to decide. It's worth noting that a lot of people run MySQL/InnoDB the same way, and there are lots of bugs and special behaviors that end up making other isolation levels unusable for various reasons when various features are used in MySQL.

I think Mark Callaghan's tweet above is largely true. But even the PostgreSQL docs, as clear as they are, have some things that are hard to parse. Does the first part of this excerpt contradict the second part? (Emphasis mine):

> a SELECT query (without a FOR UPDATE/SHARE clause) sees only data committed before the query began; it *never sees either uncommitted data or changes committed during query execution by concurrent transactions*. In effect, a SELECT query sees a snapshot of the database as of the instant the query begins to run. However, SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that *two successive SELECT commands*

> *can see different data, even though they are within a single transaction, if other transactions commit changes during execution of the first SELECT.*

Even PostgreSQL's apparently less complicated MVCC implementation has thorny questions such as those. On more careful reading, the meaning becomes clear (and I don't see how to improve it, by the way). The issue remains: these are subtle topics that inherently require close attention to detail.

One of the most elegantly put points in this documentation page is the remark that "Consistent use of Serializable transactions can simplify development."

It's Not Just MySQL And PostgreSQL

Many other systems implement some type of MVCC. All of them, as per the name, rely on multiple versions of records/rows, and deal with the various conflicts between these multiple versions in various ways. Some more complex, some less. The behavior the developer sees is heavily influenced by the underlying implementation.

And developers have to deal with this. If you're going to use one of these systems competently, you must know the intricacies. I saw this again and again while consulting with MySQL users. Many developers, including myself, have written applications that fall afoul of the MVCC implementation and rules. The results?

- Performance problems.
- Availability problems.
- Deadlocks and other errors.
- Bugs. Horrible, subtle bugs in the way the app uses the database.

The only systems I'm aware of that can avoid these problems are those that use strategies such as single-writer designs. These, contrary to what their proponents will say about them, generally do not scale well at all. Many a MyISAM has been reinvented by database developers who don't understand why MyISAM doesn't scale.

Back To Eventual Consistency

In contrast with that nightmare of complexity, I'm not so sure eventual consistency is really all that hard for developers to deal with. The developers will *always* need to be aware of the exact behavior of the implementation they're writing against, relational or not. I've studied quite a few eventually consistent databases (although I'll admit I've spent most of my career elbows deep in InnoDB) and it seems hard to believe Cassandra or Riak is really more complex to develop against than InnoDB, for the use cases that they serve well.

Eventually consistent is easy to ridicule, though. Here's one of my favorites:

Stewart Smith
@stewartsmith

Eventually consistent #FiveWordTechHorrors

(If you don't get the joke, just wait a while. It'll come to you.)

Can we have the best of all worlds? Can we have transactional behavior with strong ACID properties, high concurrency, etc, etc? Some claim that we can. FoundationDB, for example, asserts that it's possible and that their implementation is fully serializable, calling other isolation levels weak, i.e. not true I-as-in-ACID. I haven't yet used FoundationDB so I can't comment, though I have always been impressed with what I've read from them.

But since I am not ready to assert that there's a distributed system I know to be better and simpler than eventually consistent datastores, and since I certainly know that InnoDB's MVCC implementation is full of complexities, for right now I am probably in the same position most of my readers are: the two viable choices seem to be single-node MVCC and multi-node eventual consistency. And I don't think MVCC is the simpler paradigm of the two.

References

Further reading:

- Adrian Colyer on quantifying isolation anomalies
- Kyle Kingsbury on Galera Cluster

---

1. If you don't tweet me puns and acid-cat meme pictures about this paragraph, I shall be disappointed in you.
2. Pun intended.
3. Also note that PostgreSQL used to provide only *two* isolation levels, and the documentation used to make the same comment about it being the only sensible thing to do. It's not quite clear to me whether this is meant to imply that it's the only sensible way to implement MVCC, or the only sensible way to implement PostgreSQL's MVCC.