READ MODE · ANSWER FIRST

Linearizability vs Serializability vs Snapshot Isolation

An interactive, visual explanation of Linearizability vs Serializability vs Snapshot Isolation, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

The whole story in 6 lines

Linearizability vs Serializability vs Snapshot Isolation becomes easier to reason about when every stage is connected as one system.

1. A consistency model is a contract between the database and the client. It says: here are the outcomes I will give you and here are the...
2. Linearizability is about one key at a time. It promises that once a write completes, every subsequent read sees that value or a newer one.
3. Serializability lets the database interleave transactions for performance, as long as the final result matches some serial schedule.
4. Strict serializability is the gold standard. It is the multi-key analogue of linearizability. Spanner gives you this.
5. Write skew is the canonical SI anomaly. Two transactions read the same data, both pass an invariant check, both write different keys...
6. Picking a model is a tradeoff. Stronger means safer but slower and harder to scale.

STAGE 1 OF 8

Setup

Welcome. The words linearizability, serializability, snapshot isolation get used as if they mean the same thing. They do not. Each is a promise about what your database is allowed to show you. Let us start by getting the vocabulary right.

An operation is a single read or write of one key. A transaction is a group of operations that should logically happen together. The difference matters: linearizability talks about operations, serializability talks about transactions.

An anomaly is any outcome the database produces that should not be possible if everything ran cleanly one at a time. Anomalies are how we measure consistency models. A model is defined by which anomalies it forbids.

Real-time order is the wall clock. If transaction A finishes before transaction B starts, A came first in real time. Some models care about this, some do not.

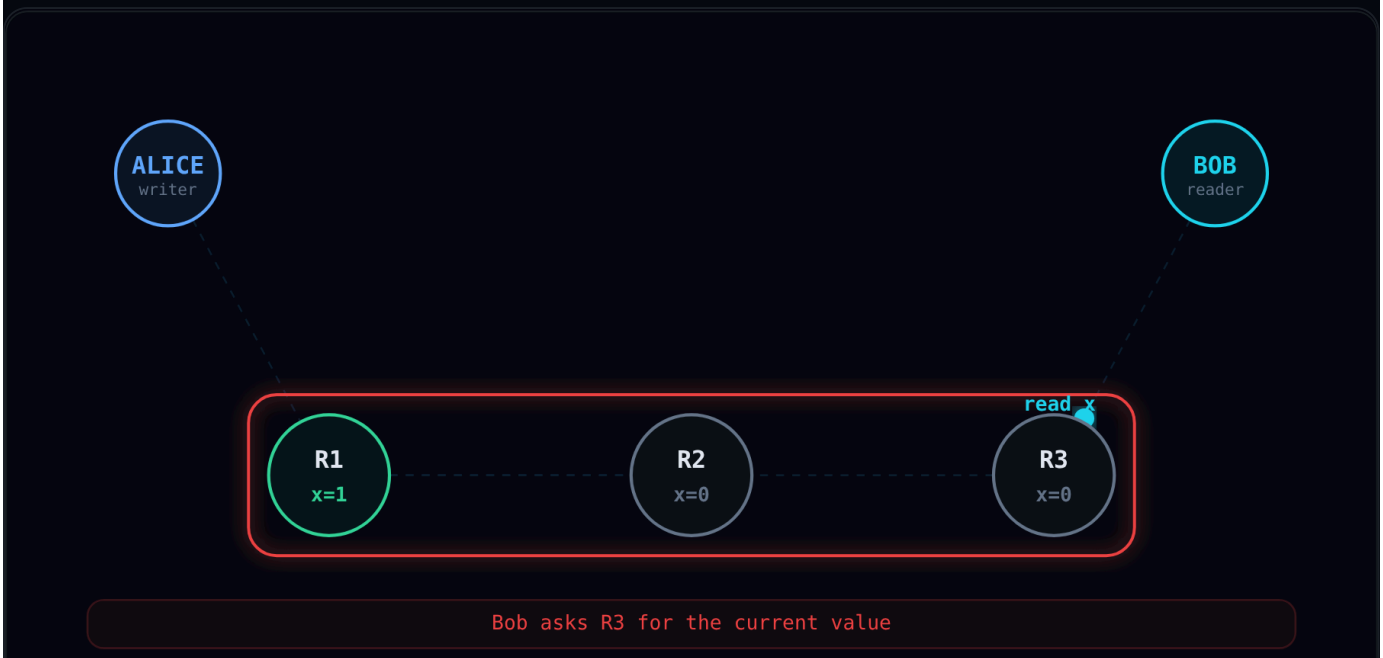
Serial order is the order the system pretends things happened in, regardless of actual interleaving. Two transactions might overlap in real time but the database can still claim a serial order where one came before the other.

An invariant is a rule the data must always obey. Account balances cannot go negative. At least one doctor must be on call. Snapshot isolation can break invariants in ways that look fine for each transaction in isolation. We will see this on Stage 5.

Over the next six stages we will build up the picture: why models exist, then linearizability, serializability, strict serializability, snapshot isolation, and finally the lattice that ranks them all. Let us begin with the most basic question: what goes wrong without rules?

STAGE 2 OF 8

Why models exist



02 WHY MODELS EXIST

Here is the setup. There is one logical key x. The database stores three copies on three replicas. Alice and Bob are two clients sending requests. They have no idea where their requests land.

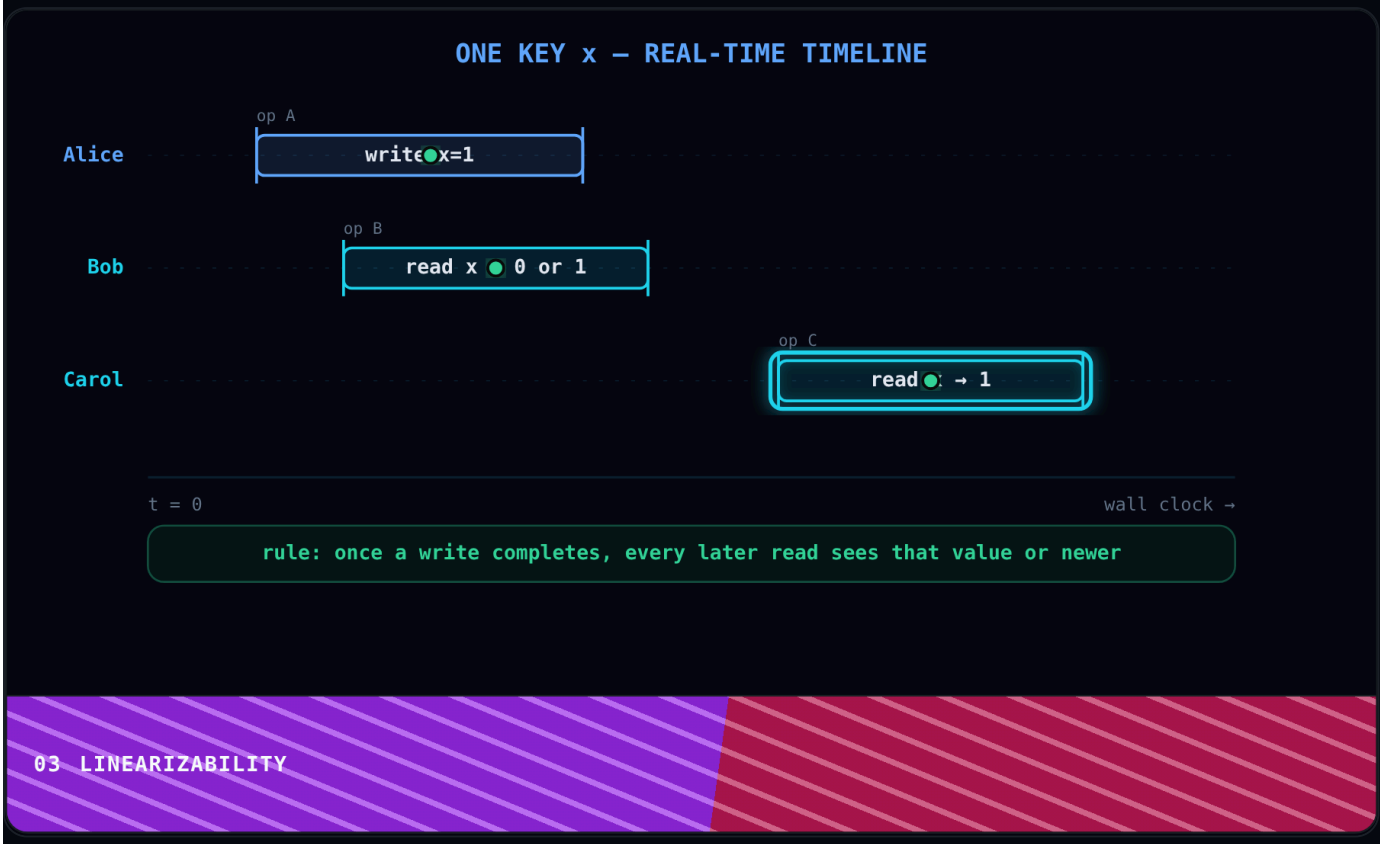
Alice writes x equals 1 to replica R1. R1 acknowledges. As far as Alice knows, the write is done. The system told her so.

A moment later, Bob reads x from replica R3. R3 has not received the new value yet. So Bob gets x equals 0. The old value. After Alice already heard the write succeeded.

This is a stale read anomaly. Switch the Replication control in the sidebar to Sync. The write now waits for every replica before acknowledging, and Bob always sees the new value. That is the contract changing under your feet.

Without a written rule, the database is allowed to give either answer. Stale reads, lost writes, ghost values, all of it is legal until you pick a model that forbids it. That is what a consistency model does. The next four stages each forbid something different.

Linearizability



03 LINEARIZABILITY

We just saw that without rules, Bob can read stale values. Linearizability is the rule that fixes this for a single key. Look at the timeline. Three operations on key x. Time flows left to right.

Operation A is Alice writing x equals 1. The bar shows when Alice sent the request and when she got the acknowledgment. The dot inside the bar is the linearization point: the moment the write actually took effect.

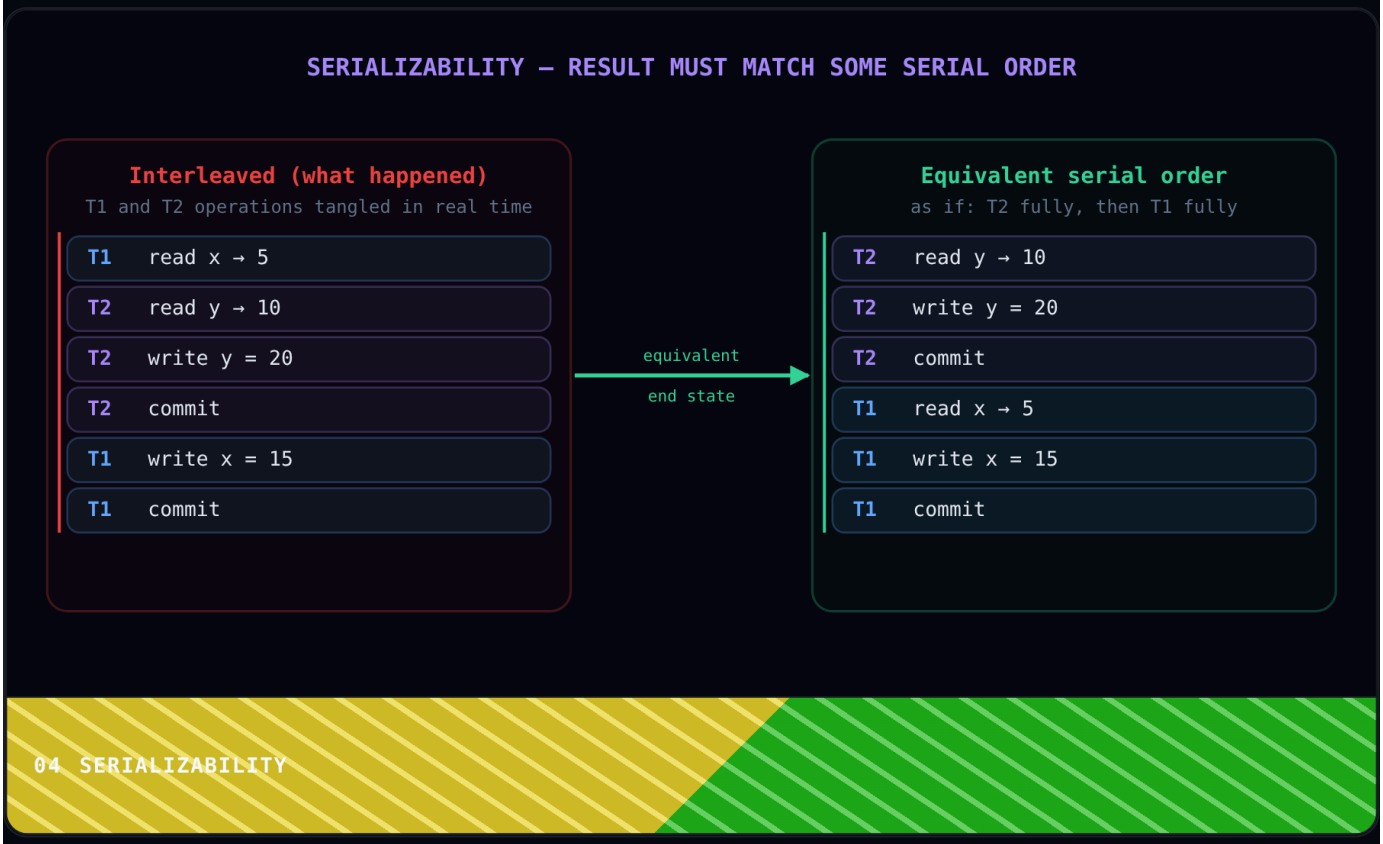
Operation B is a read by Bob that overlaps Alice's write in real time. Because the operations overlap, the database has freedom. Bob is allowed to see either 0 or 1. Both are linearizable.

Operation C is another read, but it starts after Alice's write completed. Try the Read C control. Set it to After. Now C must return 1. Returning 0 would mean the database forgot the write, which violates real-time order.

That is the rule in one line: once a write completes, every read that starts later must see that value or a newer one. The system behaves as if there were one global copy and operations took turns touching it.

But linearizability only talks about one key at a time. It says nothing about what happens when a transaction touches multiple keys. For that we need a different model. Next up: serializability.

Serializability



04 SERIALIZABILITY

We just learned linearizability for one key. But real applications do transactions: groups of operations that should hang together. Look at T1 and T2 on the left. Each touches two keys.

In the actual execution, the operations are interleaved. T1 reads x. T2 reads y. T2 writes y. T1 writes x. Reads and writes are tangled across both transactions.

Serializability asks: is this tangle equivalent to running the transactions one after another in some order? If yes, the schedule is serializable. The database is allowed to do this.

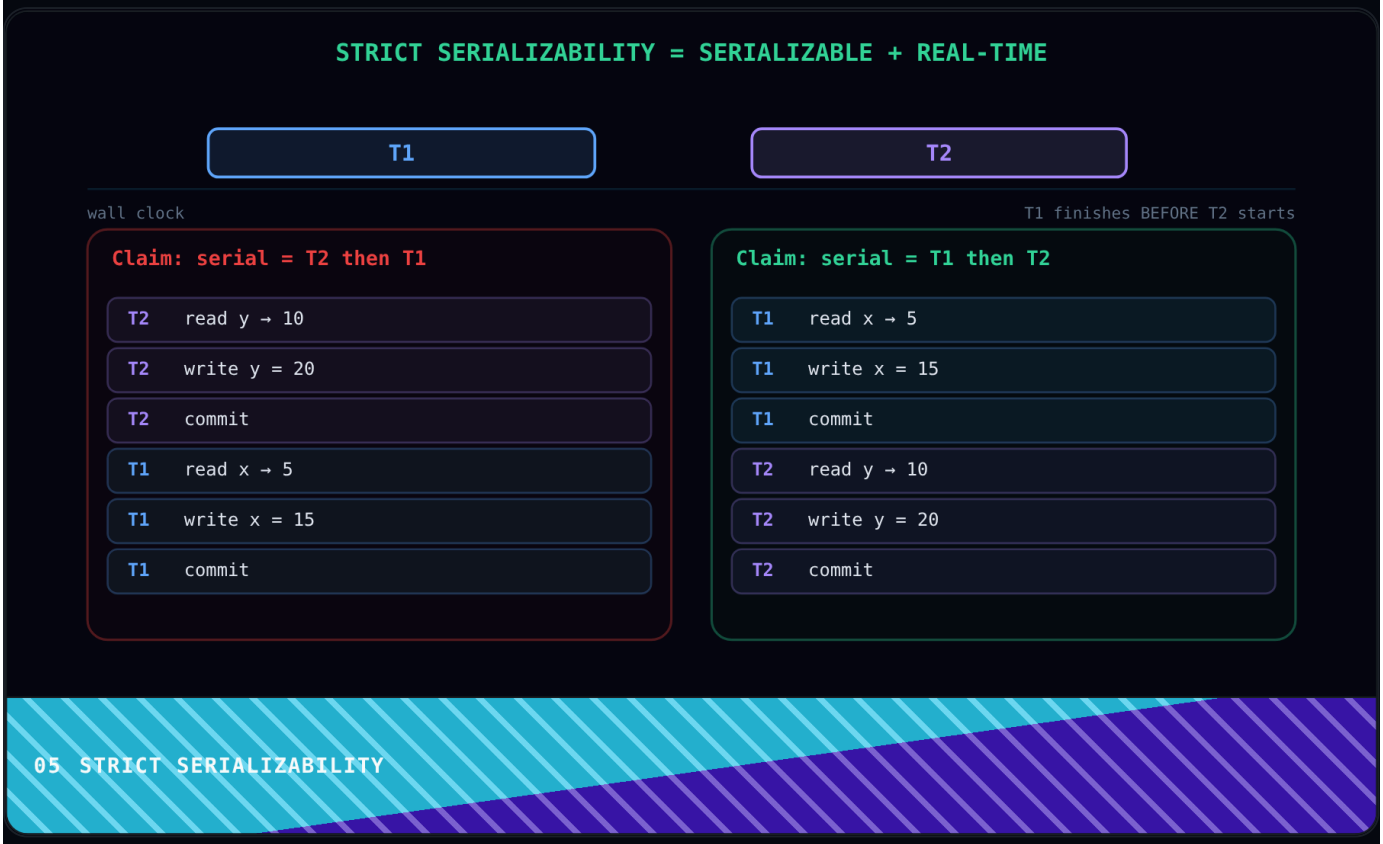
On the right we show the equivalent serial order: T2 ran fully, then T1 ran fully. Every key ends up with the same value as the interleaved version. Try the Serial order control to see the other valid claim.

Here is the catch. Notice that T1 actually started first in real time. But the database is claiming T2 happened first. That is allowed. Serializability does not require the serial order to match the wall clock.

So serializability protects against multi-key anomalies inside transactions, but it can still violate real-time ordering across transactions. To get both, we need a stronger model. That is strict serializability. Next stage.

STAGE 5 OF 8

Strict Serializability



05 STRICT SERIALIZABILITY

We just saw serializability allows the database to reorder transactions across the wall clock. That feels wrong if you watched T1 commit before T2 even started. Strict serializability is the fix.

Look at the wall clock at the top. T1 starts, runs, commits. Then T2 starts, runs, commits. In real time, T1 is clearly first. No overlap.

Plain serializability would let the database claim the equivalent serial order is T2 then T1, as long as the end state matches. That is the schedule on the left. Legal under serializable.

Strict serializability adds one more rule: if T1 committed before T2 began, the serial order must put T1 first. The schedule on the right is the only valid one. Switch the Model control to compare.

You can think of strict serializability as serializability plus linearizability. Multi-key transactions that also respect real-time order. This is what Google Spanner promises with its TrueTime clocks.

But strict serializability is expensive. Most databases settle for something weaker. The most popular weaker model is snapshot isolation, and it has a famous failure mode. We see it next.

STAGE 6 OF 8

Snapshot Isolation

06 SNAPSHOT ISOLATION

The setup is the doctors-on-call problem. Hospital rule: at least one doctor must stay on call. Right now Alice and Bob are both on call. Two doctors, rule satisfied.

Alice and Bob both want to take themselves off call. They each open a transaction at the same time. SI gives each transaction its own frozen snapshot of the database.

Alice's transaction reads the on-call list. She sees two doctors: herself and Bob. She thinks: it is safe to take myself off, Bob will still be on call.

Bob's transaction reads the on-call list at the same moment. He sees two doctors: himself and Alice. He thinks the same thing: it is safe to take myself off, Alice will still be on call.

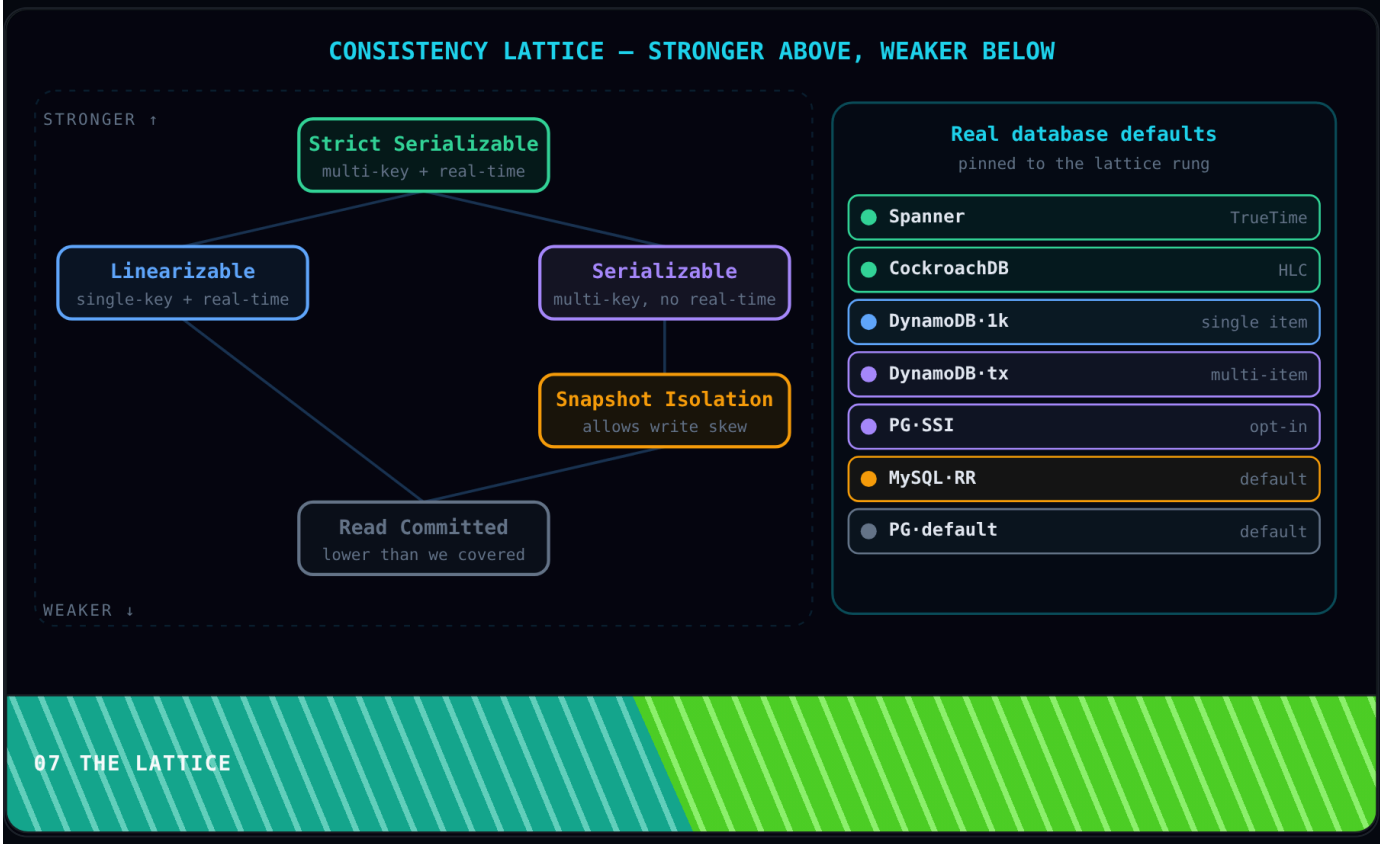
Alice writes alice.on_call equals false. Bob writes bob.on_call equals false. Different keys. SI's first-committer-wins rule only catches conflicts on the same key. Both transactions commit successfully.

Now read the database. Zero doctors on call. The invariant is broken. Neither transaction did anything wrong by itself. The bug only exists in the gap between their snapshots and the real world. Switch the Mode control to Serializable to see the schedule that would prevent this.

This is write skew. It is why people who say snapshot isolation is close enough to serializable are wrong. Same-key conflicts are caught. Cross-key invariants are not. Now let us put all four models on a single map.

STAGE 7 OF 8

The lattice



07 THE LATTICE

Here is the lattice. At the top sits Strict Serializability: multi-key transactions plus real-time order. Below it, two siblings. Linearizability cares about real-time order on single keys. Serializability cares about transactions but ignores real-time order. They forbid different things.

Below Serializability sits Snapshot Isolation. SI forbids most anomalies but not write skew. Below SI are even weaker levels like Read Committed and Read Uncommitted that we did not even cover, because they forbid almost nothing.

Now look at the pins. Google Spanner sits at Strict Serializability thanks to TrueTime atomic clocks bounding clock skew. CockroachDB also markets serializable by default with hybrid logical clocks for ordering.

PostgreSQL's default is Read Committed, weaker than everything we covered. You can opt in to Serializable Snapshot Isolation, but you have to ask for it. Most production Postgres is at Read Committed.

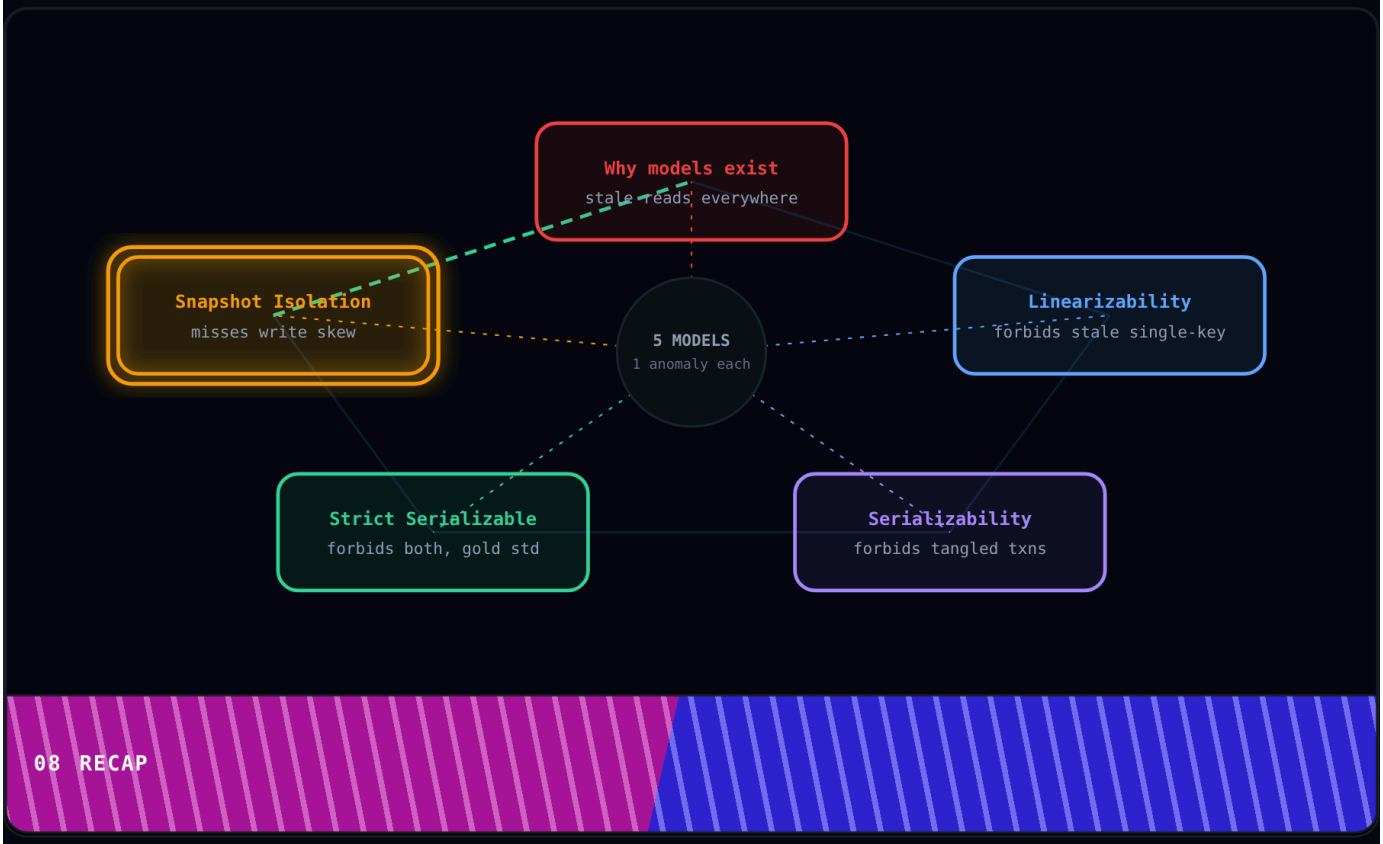
MySQL InnoDB defaults to Repeatable Read, which is roughly snapshot isolation. DynamoDB single-item writes are linearizable. Multi-item transactions are serializable. Cassandra without lightweight transactions gives you eventual consistency, off the bottom of this map.

Use the Highlight control to focus on one model and see which systems ship it as default. The lesson: always check what your database promises by default. The marketing word ACID says nothing about which level you actually got.

You now have the full picture. Let us zoom out and tie it all together in the recap.

STAGE 8 OF 8

Recap



08 RECAP

We started by asking why isolation models exist. Without rules, two clients reading the same key can see different values. The replicas drift, and every weird answer is technically allowed.

Linearizability fixed that for a single key. Once a write completes, every later read must see that value or a newer one. The system pretends one global copy exists. Single-key, real-time.

Serializability handled multi-key transactions. The interleaved execution must be equivalent to some serial order. But it does not require that order to match the wall clock. Multi-key, no real-time.

Strict Serializability combined both. Multi-key transactions plus real-time order. The gold standard. Spanner-grade. The price is coordination, often using physical clocks bounded by hardware.

Snapshot Isolation gave us a popular weaker model. Each transaction reads from a snapshot. Conflicts on the same key abort. But cross-key invariants can break, the doctors-on-call write skew problem.

Together these models form a lattice. Stronger means more forbidden and more expensive. Weaker means faster but the application must reason about more anomalies. Look at all five together. Each one defeats a different kind of bug.

The takeaway in one sentence: linearizability is about single-key real-time order, serializability is about multi-key serial equivalence, strict serializability combines both, and snapshot isolation is the popular shortcut that misses write skew. You will never confuse these terms again.