April 2025 - PGDay Chicago

Implementing **Strict Serializability** w/ **pg_xact**

Jimmy Zelinskie, AuthZed

Agenda

Structure for this presentation

1. Agenda & Bio
2. Consistency 101
3. Strict Serializability
4. Who needs Strict Serializability?
5. Naive Approach
6. pg_xact approach

Who is **Jimmy Zelinskie**

Jimmy Zelinskie is a software engineer and product leader with the goal of empowering the world through the democratization of software through open source development. He's currently the CPO and cofounder of authzed where he's focused on bringing hyperscaler best-practices in authorization software to the industry at large.

At CoreOS, he helped pioneer the cloud-native ecosystem by starting and contributing to many of its foundational open source projects. After being acquired by Red Hat, his focus shifted to the enablement and adoption of cloud-native technologies by mature enterprise stakeholders. To this day, he still contributes to cloud-native ecosystem by building the future on top of these technologies and maintaining standards such as Open Container Initiative (OCI).

authzed

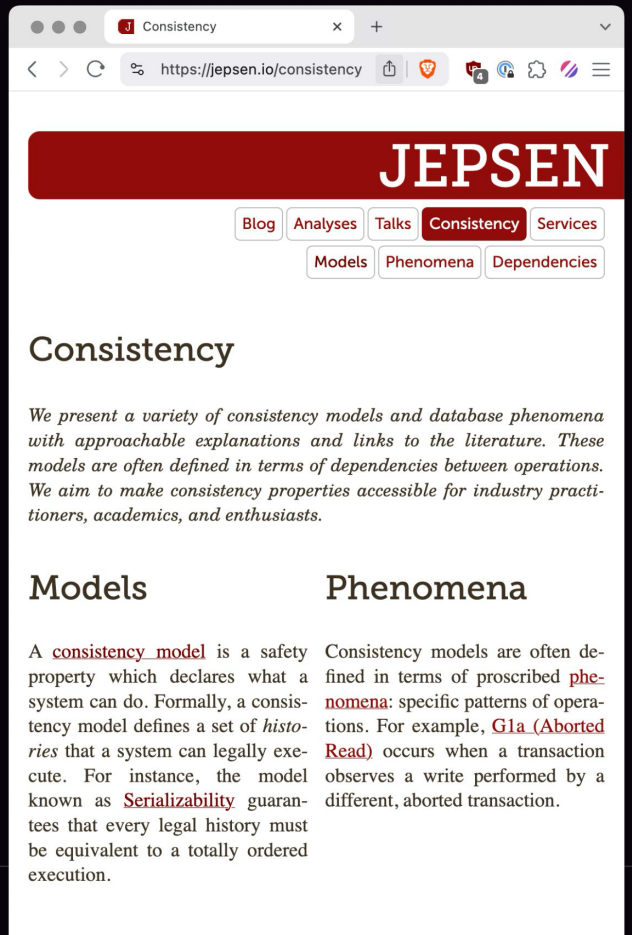
CoreOS

What is **consistency**

You might be familiar with the acronym ACID. Consistency is the C in ACID and is one of the critical properties for correctness in any system managing state.

While there are formal definitions, I prefer a colloquial definition of consistency: the **contract for how data can be observed**.

Your applications are built with assumptions about the data it's using and when those assumptions are incorrect, actions can take place that shouldn't and critical data can become corrupt. If your state management system cannot match your application's requirements, you will have to re-architect.



Give me an **example**

An e-commerce platform is a really intuitive example for demonstrating consistency without mentioned software or servers at all.

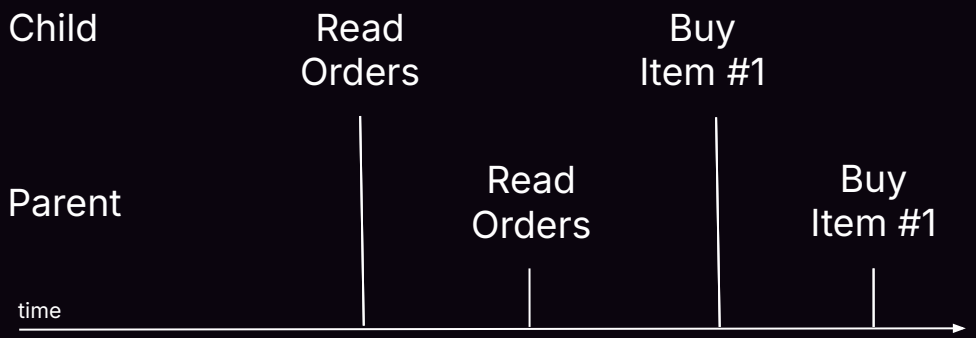
Consider a parent supervising the online purchase made by a child; if the child does not purchase the item, the parent will.

This depiction on the right demonstrates the ideal scenario: the child checks the orders, sees the item hasn't yet been purchased, purchases the item, and the parent checks the orders to confirm the child made the purchase.

Give me an **example**

What's actually far more likely than the ideal case are scenarios where these events are ordered in surprising ways.

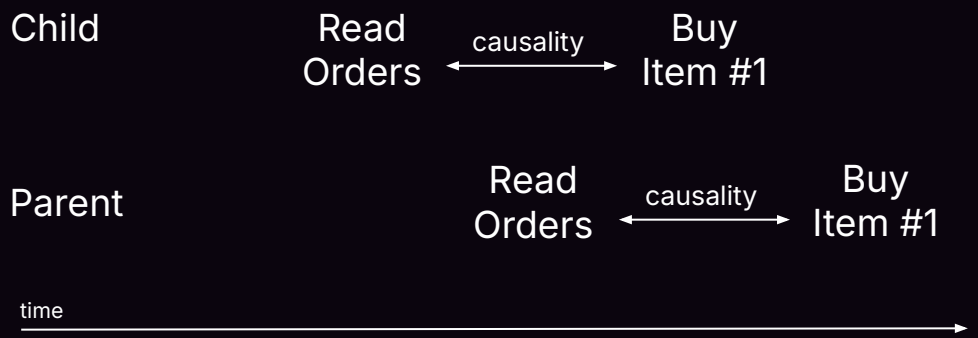
Consider if the parent reads the orders after the child has read the orders, but before they have purchased. An unintended double-purchase takes place.



Give me an **example**

The previous example is surprising because the purchase **causally depends** on the reading of the orders.

One solution is to simply merge any causal dependencies into one single, atomic action. That would focus on **atomicity** and not consistency. These terms are distinct, but so closely related that many scenarios will discuss both such that it may be pedantic to focus on the difference.



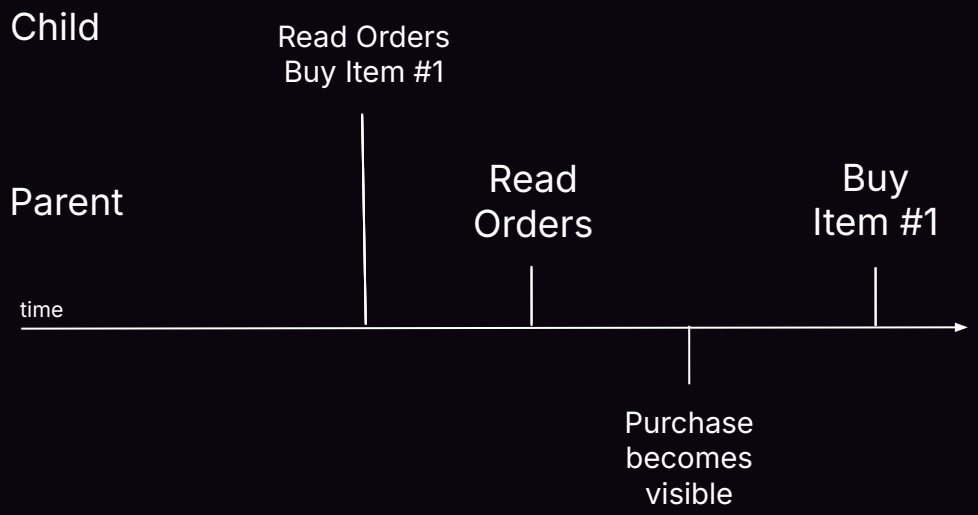
Give me an **example**

However, not all scenarios can be fixed by considering atomicity.

Consider the child atomically performed their read and purchase, but it took an amount of time before that purchase became visible.

Now, the parent could read in that time before the purchase is visible and still cause a double-purchase.

This time until a change becomes visible is the crux of the difficulty for implementing Strict Serializability.



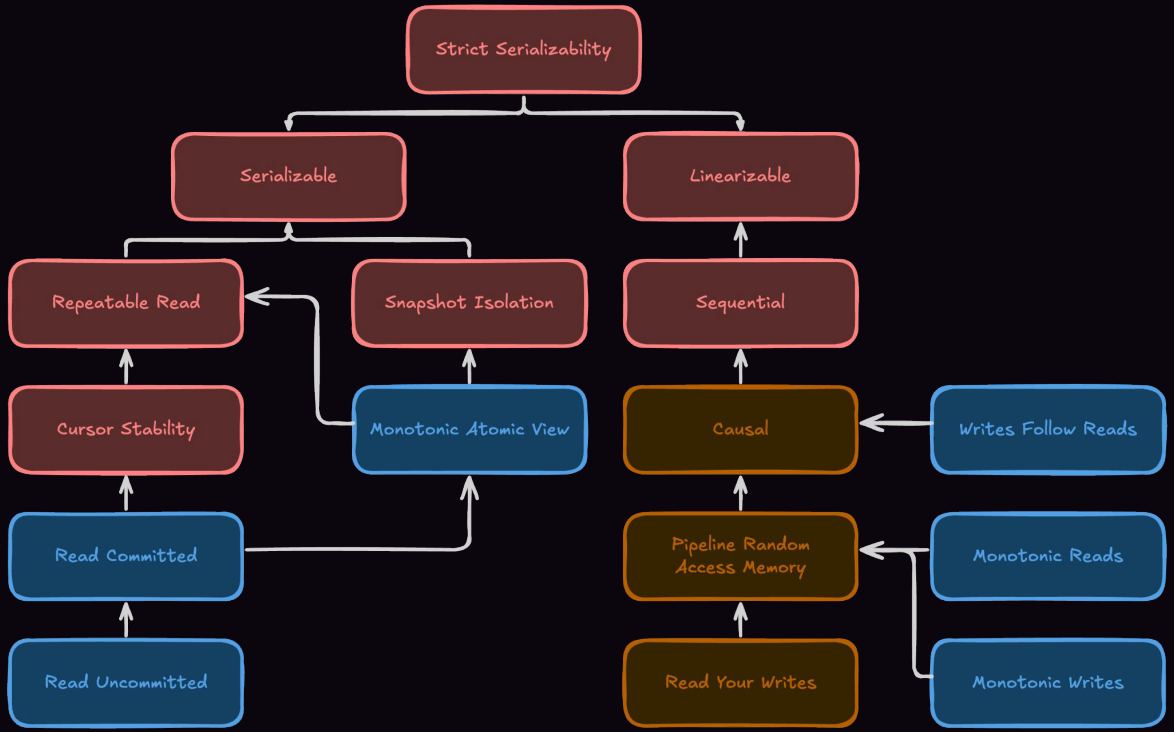
Red: Usually cannot tolerate network failures
Orange: Tolerates sticking to existing non-faulty nodes
Blue: Tolerates on every non-faulty node

What is **Strict Serializability**

Informally, Strict Serializability means that operations appear to have occurred in some order, consistent with the real-time ordering of those operations (e.g. if operation A completes before operation B begins, then A should appear to precede B in the serialization order).

Strict serializability implies serializability and linearizability. You can think of strict serializability as serializability total order of transactional multi-object operations, plus linearizability's real-time constraints.

Alternatively, you can think of a strict serializable database as a linearizable object in which the object's state is the entire database.



https://jepsen.io/consistency/models
https://jepsen.io/consistency/models/strong-serializable

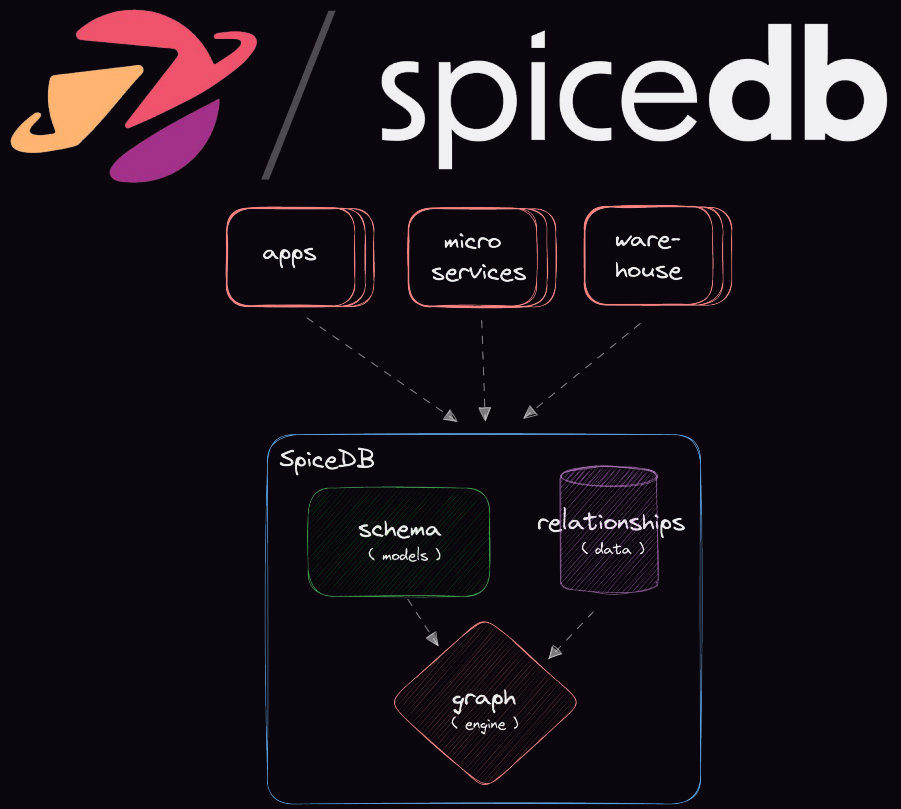
Why do we need **Strict Serializability**

SpiceDB is a domain-specific database for storing and querying authorization data. It has a pluggable storage backend that includes and implementation using Postgres. This is similar to how relational databases usually sit on top of a pluggable storage engine often implemented with key-value store libraries like RocksDB.

SpiceDB's Postgres storage implementation intends to support all cloud platform's managed Postgres services. This means no custom plugins -- only extensions that are available by default everywhere.

Without Strict Serializability, authorization systems suffer from a vulnerability called the **New Enemy Problem**. In order to be secure, SpiceDB must achieve this level of consistency in the implementation of each storage backend.

Most importantly, SpiceDB offers a Watch API that streams an totally ordered changefeed of updates to a consumer.



What's the **naive approach**

The highest isolation level configurable in Postgres is Serializable. Serializability does not impose any real-time, or even per-process constraints. If process A completes write w, then process B begins a read r, r is not necessarily guaranteed to observe w.

Our first attempt was to build our own MVCC layer on top of Postgres.

This works, but is SLOW. How can we make this faster? By reusing the work that the Postgres MVCC is already doing!

```
CREATE TABLE relation_tuple_transaction (
    id BIGSERIAL NOT NULL,
    timestamp TIMESTAMP WITHOUT TIME ZONE
        DEFAULT now() NOT NULL,
    CONSTRAINT ...,
);

CREATE TABLE relation_tuple (
    namespace VARCHAR NOT NULL,
    object_id VARCHAR NOT NULL,
    relation VARCHAR NOT NULL,
    userset_namespace VARCHAR NOT NULL,
    userset_object_id VARCHAR NOT NULL,
    userset_relation VARCHAR NOT NULL,
    created_transaction BIGINT NOT NULL,
    deleted_transaction BIGINT NOT NULL DEFAULT
'9223372036854775807',
    CONSTRAINT ...,
);
```

Introducing **pg_xact**

Introduced in Postgres 7 as pg_clog; renamed to pg_xact in Postgres 10. Available on all cloud services.

Formally, the name of a transaction metadata directory on disk; set of functions for access transaction metadata in queries; most importantly snapshots and xids (transaction IDs).

Snapshots encompasse a range of transactions. A transaction isn't live/ordered until it is no longer marked as in-progress in a snapshot.

snapshot == (min, max, [in-progress])

```
postgres=# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 172154058:172154058:

postgres=# select pg_snapshot_xmin(pg_current_snapshot());
 pg_snapshot_xmin
------------------
        172154058

postgres=# select pg_snapshot_xmax(pg_current_snapshot());
 pg_snapshot_xmax
------------------
        172154058

postgres=# select pg_snapshot_xip(pg_current_snapshot());
 pg_snapshot_xip
-----------------

postgres=# select pg_visible_in_snapshot('172154057'::xid8,
'172154058:172154058:');
 pg_visible_in_snapshot
------------------------
 t
```

An xid-aware **schema**

```
Table "public.relation_tuple_transaction"
  Column   |            Type             | Collation | Nullable |             Default              |
-----------+-----------------------------+-----------+----------+----------------------------------+
 timestamp | timestamp without time zone |           | not null | (now() AT TIME ZONE 'UTC'::text) |
 xid       | xid8                        |           | not null | pg_current_xact_id()             |
 snapshot  | pg_snapshot                 |           | not null | pg_current_snapshot()            |
 metadata  | jsonb                       |           | not null | '{}'::jsonb                      |
```

```
Table "public.relation_tuple"
     Column       |          Type          | Collation | Nullable |           Default
|-----------------+------------------------+-----------+----------+-----------------------------+
namespace         | character varying      |           | not null |                             |
object_id         | character varying      |           | not null |                             |
relation          | character varying      |           | not null |                             |
userset_namespace | character varying      |           | not null |                             |
userset_object_id | character varying      |           | not null |                             |
userset_relation  | character varying      |           | not null |                             |
created_xid       | xid8                   |           | not null | pg_current_xact_id()        |
deleted_xid       | xid8                   |           | not null | '9223372036854775807'::xid8 |
```

Querying **alive transactions**

A SpiceDB revision is a synthetic Postgres snapshot created by putting a transaction row's xid into the completed section of the row's snapshot. For scenarios when we need the freshest visible data, we can simply use pg_current_snapshot().

Once we have a snapshot, we can rely on WHERE pg_is_visible_in_snapshot(...) to do the heavy lifting when filtering for living tuples.

SpiceDB also has an API for emitting an ordered changefeed; this is where ordering is most necessary. pg_xact has one last trick up it's sleeve, the ability to read the timestamp metadata from the WAL of when a transaction was written. This provides us our total ordering.

```
// Psuedocode for marking an xid as completed
if txid < xmin return
if txid > xmax append [xmax, txid) to xip & set xmax to txid
if txid is in xip, delete it
if len(xip) > 0 xmin = xip[0] else xmin=xmax
```

```
-- Select all of the revisions AFTER snapshot $1
SELECT xid, snapshot, timestamp
FROM
  relation_tuple_transaction
WHERE
  xid ⩾ pg_snapshot_xmax($1)
  OR (
    xid ⩾ pg_snapshot_xmin($1)
    AND NOT pg_visible_in_snapshot(xid, $1)
  )
ORDER BY
  pg_xact_commit_timestamp(xid :: xid),
  xid;
```

What's the **catch**

One of our customers has relied on this code on a workload running on the largest Postgres instance on Azure. However, this solution is not without tradeoffs.

The counter for transaction IDs are independent per Postgres deployment. While this works fine for replication, if you want to copy tables into a new Postgres, you need to run a script to increment the transaction counter to the same place as the original instance.

Xids can overflow, but Postgres has logic to "freeze" sufficiently old txids so long as VACUUM runs every 2 billion transactions. We guarantee that our SpiceDB garbage collection runs more frequently than that.

```
-- Incrementing the xid counter
DO $$
BEGIN
  FOR i IN 1..1000000 LOOP
    PERFORM pg_current_xact_id(); ROLLBACK;
  END LOOP;
END $$;
```

Thanks to

- authzed eng (Jake's design)
- postgres documentation & pgpedia
- kyle kingsbury for jepsen's consistency content
- pgday chicago organizers