Spanner: Google's Globally-Distributed Database

OSDI 2012

James C. Corbett, Jeffrey Dean, Michael Epstein, et al.

Google, Inc.

Course Presentation for COL862
Prof. Abhilash Jindal

Outline

What is Spanner?

The Core Claim

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database.

- It is the **first system** to distribute data at a global scale and support **externally-consistent distributed transactions**.
- It shards data across many sets of Paxos state machines, spread across datacenters worldwide.
- Provides features like:
  - Automatic resharding and data migration to balance load.
  - Automatic failover between replicas.
  - Fine-grained replication control by applications.

Motivation: Limitations of Prior Systems

Bigtable

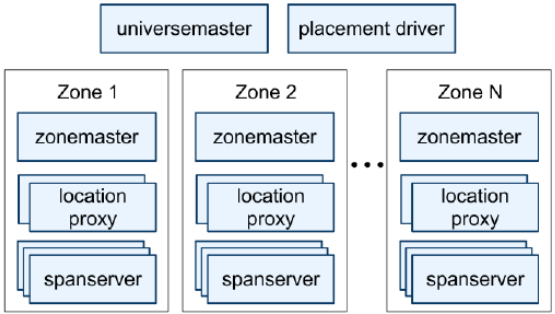
- Highly scalable key-value store.
- **Lacks:**
  - Strong consistency for wide-area replication (only eventual).
  - Cross-row transactions (a frequent complaint).

Megastore

- Provides semi-relational data model and synchronous replication.
- Popular at Google (Gmail, Picasa, etc.).
- **Lacks:**
  - Good performance.
  - **Relatively poor write throughput.**

Goal: Create a system with the **scalability** of Bigtable but with the **strong consistency** and **transactional** guarantees of a traditional database, all at a **global scale**.

High-Level Architecture



Spanner Server Organization

A Spanner deployment is a **universe**.

Universemaster A console for status monitoring and debugging.

Placement Driver Handles automated data movement between zones for load balancing and replication.

Zone The unit of administrative deployment, roughly one per datacenter.

Zonemaster Assigns data to spanservers.

Spanserver The workhorse. Serves data to clients. 100s–1000s per zone.

Spanserver Software Stack

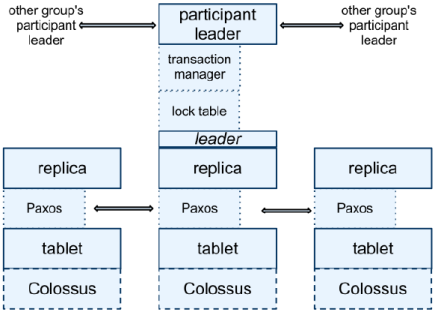
Transaction Manager Manages distributed transactions (2PC). Each leader is a "participant leader" or "coordinator leader".

Lock Table Used for concurrency control (two-phase locking) for RW transactions.

Paxos A single Paxos state machine per tablet handles replication. Uses long-lived (10s) time-based leader leases.

Tablet A bag of mappings: `(key:string, timestamp:int64) → string`. This is a key difference from Bigtable (it's multi-version).

Colossus The successor to GFS. Stores the tablet's state (B-tree files and WAL).



Spanserver Stack

Data Model: Semi-Relational

- Spanner's data model is layered on top of key-value mappings.
- It's not purely relational: **rows must have names**.
- Every table must have one or more **primary-key columns**.
- This makes it look like a key-value store where the primary key is the row's name.
- This structure lets applications control data locality by choosing their keys.
- Supports a SQL-based query language.

Transaction Types

Operation	Declared Intent	Timestamp	Concurrency
Read-Write Transaction	General transactions	Assigned at commit	**Pessimistic** (Two-Phase Locking)
Read-Only	Declared no writes before starting	System-chosen	**Lock-Free**
Snapshot Read (in the past)	Single read in the past	Client-provided	**Lock-Free**

- **Key Idea:** All transactions are assigned a **globally-meaningful timestamp** s.
- This timestamp reflects a serialization order that satisfies **external consistency** (linearizability).

Step 1: The Goal & Concurrency Problem

The Goal: Strict Serializability

We want **Serializability** (transactions are equivalent to *some* serial order) + a **Real-Time Guarantee**.

- **Invariant:** If T_1 commits *before* T_2 starts (real time), final order **must** be T_1 then T_2.

Spanner's Concurrency Model

How to handle Read-Write (RW) and Read-Only (RO) transactions?

- **RW Transactions (e.g., bank transfer):** Use **Pessimistic Concurrency Control (PCC)**.
  - Transactions acquire locks.
  - Use **Two-Phase Commit (2PC)** to coordinate locks across distributed shards.
  - Spanner's 2PC uses Paxos groups as participants/coordinators to prevent blocking on failure.
- **RO Transactions (e.g., audit):** Use **Snapshot Reads** (Lock-Free).
  - PCC is too slow for reads (would lock the whole database).

Step 2a: The Mechanism for Snapshot Reads

Multi-Version Concurrency Control (MVCC)

How Spanner supports lock-free read-only (RO) transactions:

- Read-Write (RW) txns get a **commit timestamp** s.
- All writes are **versioned** with this timestamp: $(key, s) \rightarrow value$.
- Read-Only (RO) txns get a **read timestamp** t.
- The RO txn reads the latest data version where the item's timestamp $s \leq t$.
- **Benefit:** Reads and writes don't block each other!

But this creates a new distributed problem...

Step 2b: The New Problem

The Race Condition

- An RO txn starts at $t = 5$ and asks Shard Y for data.
- A RW txn (involving Shards X and Y) is committing at $t = 2$.
- But the commit message is **delayed** by the network and hasn't reached Shard Y yet.
- Shard Y only has the old data (e.g., 'y@0') and incorrectly serves it.
- **Failure:** The read at $t = 5$ just saw "stale" data from before $t = 2$. Serializability is broken!

The solution

Use locks to ensure transactions don't interleave. We are back to Two-Pase Commits.
To avoid blocking reads, systems replace this with a *safe-time rule* (t_{safe}) that lets each shard decide when it's safe to serve reads.

Step 3a: The "Clock Skew" Problem

't_{safe}' isn't enough for Strict Serializability!

't_{safe}' ensures data is consistent, but not that the timestamps match **real-time**.

- **Scenario (Real Time = 1:00 PM):**
- T_1 (writes x) commits. Its coordinator's clock is *fast*, assigns $s_1 = 1{:}02$ PM.
- T_2 (writes y) commits. Its clock is *normal*, assigns $s_2 = 1{:}00$ PM.
- **Scenario (Real Time = 1:01 PM):**
- T_3 (RO audit) starts. It gets a read timestamp $t = 1{:}01$ PM.
- **Failure:** T_3 sees T_2's write ($s_2 \leq t$) but **misses** T_1's write ($s_1 > t$).

This is a disaster! T_3 started *after* T_1 finished but didn't see its data.

Step 3b: The "Secret Sauce": TrueTime

The Problem

In a global system, how do you know if event A in Datacenter 1 happened *before* event B in Datacenter 2? Simple timestamps aren't enough due to clock skew.

TrueTime: The Solution

A new API that **directly exposes clock uncertainty**.

Method	Returns
`TT.now()`	`[earliest, latest]` (a *TTinterval*)
`TT.after(t)`	true if time `t` has definitely passed
`TT.before(t)`	true if time `t` has definitely not arrived

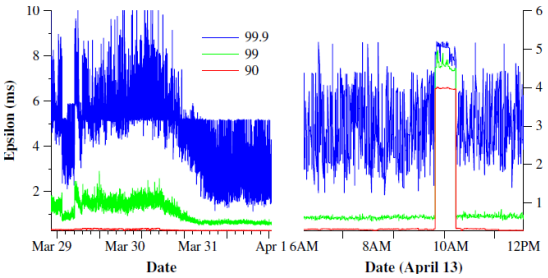
- `TT.now()` is guaranteed to return an interval that contains the absolute time of the invocation.
- Let ϵ be the clock uncertainty. If ϵ is large, Spanner **waits** for the uncertainty to pass.

TrueTime: Example & Implementation

Implementation

- **Time Masters** per datacenter.
  - Majority have GPS receivers.
  - Some have atomic clocks ("Armageddon masters").
- **Timeslave Daemon** per machine.
  - Polls multiple masters.
  - Uses Marzullo's algorithm to reject liars.
  - ϵ increases (drifts) between polls.

- ϵ is a "sawtooth" function, typically 1-7ms.
- The average $\bar{\epsilon}$ is $\sim$4ms.
- Spikes can occur due to network congestion or master failures.



Distribution of ϵ

Marzullo's Algorithm: Overview

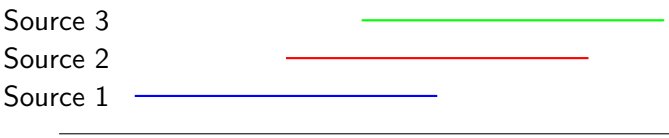
The Problem

Each Timeslave daemon polls multiple Time Masters (GPS or atomic clocks) and receives intervals $[t_{min}, t_{max}]$ representing each master's estimate of the current absolute time. Due to network delay, failures, or clock drift, intervals may not perfectly overlap.

The Goal

Determine the **most likely interval** that contains the true absolute time.

- Marzullo's algorithm finds the interval with the maximum overlap among all reported intervals.
- This interval defines $TT.now() = [t_{earliest}, t_{latest}]$ used by Spanner for external consistency.



Marzullo's Algorithm: Step-by-Step

Intuition

By picking the interval where the most sources agree, Spanner can safely assume the absolute time lies within that range despite network delays or faulty clocks.

1. Collect n intervals $[t_i^{min}, t_i^{max}]$ from n time sources.
2. Sort all interval endpoints (both min and max) in increasing order.
3. Initialize an `overlap_count` to zero.
4. Sweep through sorted endpoints:
   - If it's a `start` of an interval, increment `overlap_count`.
   - If it's an `end` of an interval, decrement `overlap_count`.
   - Keep track of the range where `overlap_count` is maximal.
5. The range with maximum overlap is returned as $[t_{earliest}, t_{latest}]$.

Guaranteeing External Consistency

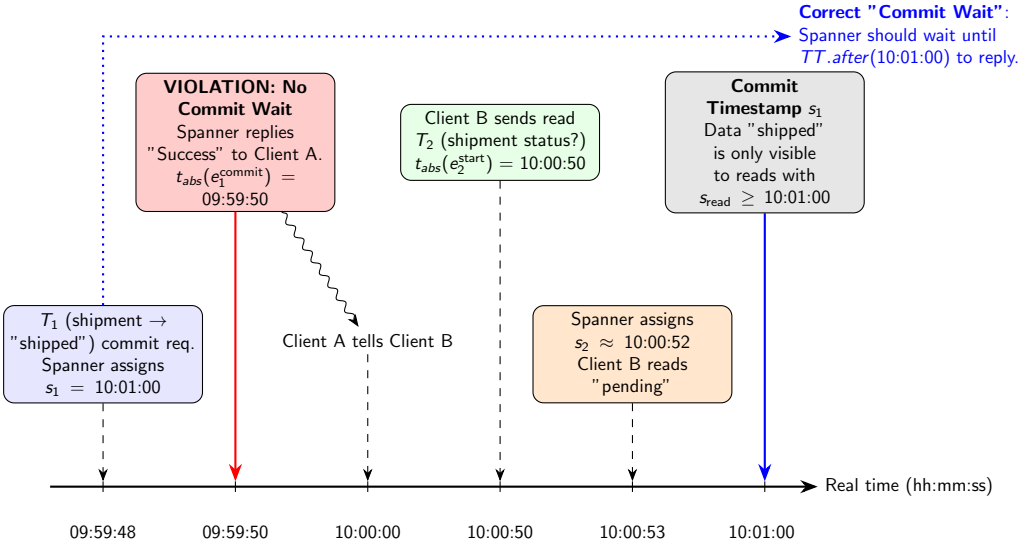
The Invariant

If a transaction T_1 commits *before* another transaction T_2 starts, then T_1's commit timestamp (s_1) must be smaller than T_2's commit timestamp (s_2).
Formally: $t_{abs}(s_1^{commit}) < t_{abs}(s_2^{start}) \implies s_1 < s_2$

How? Spanner enforces two rules for RW transactions:

1. **Start Rule:** The coordinator leader for T_i assigns a commit timestamp $s_i \geq TT.now().latest$, computed *after* receiving the prepare acknowledgements.
2. **Commit Wait Rule:** The coordinator **must wait** until `TT.after(`s_i`)` is true before allowing clients to see the committed data. (i.e., $TT.now().earliest > s_i$)

Stale Read Due to Missing Commit-Wait



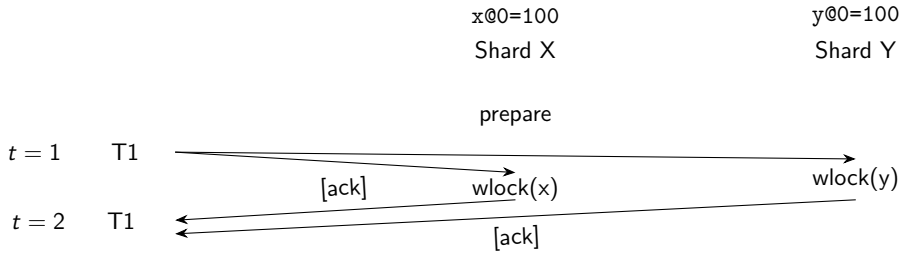
Serving Reads: The t_{safe} Concept

- For lock-free snapshot reads (at timestamp t), how does a replica know it has all data up to t?
- **Safe Time (t_{safe}):** The maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at t if $t \leq t_{safe}$.
- $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$

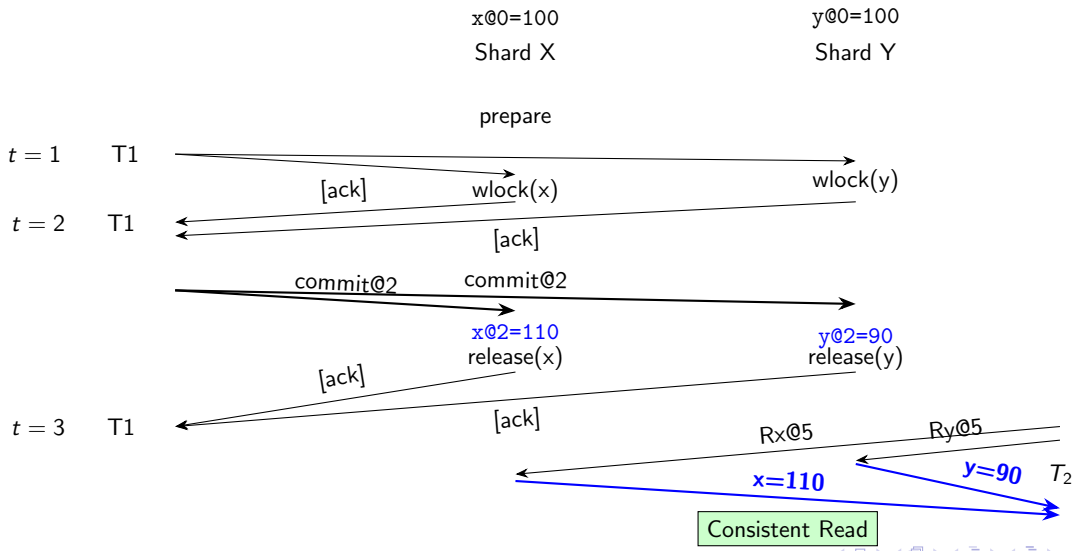
  t_{safe}^{Paxos} The timestamp of the highest-applied Paxos write. Since writes are applied in order, no more writes will occur $\leq t_{safe}^{Paxos}$.

  t_{safe}^{TM} The "Transaction Manager" safe time.
  - If there are **zero prepared** 2PC transactions, $t_{safe}^{TM} = \infty$.
  - If there are prepared transactions, their outcome is unknown. They might commit at some timestamp $s_{commit} \geq s_{prepare}$.
  - Therefore, $t_{safe}^{TM} = \min(\text{all } s_{prepare} \text{ timestamps}) - 1$.

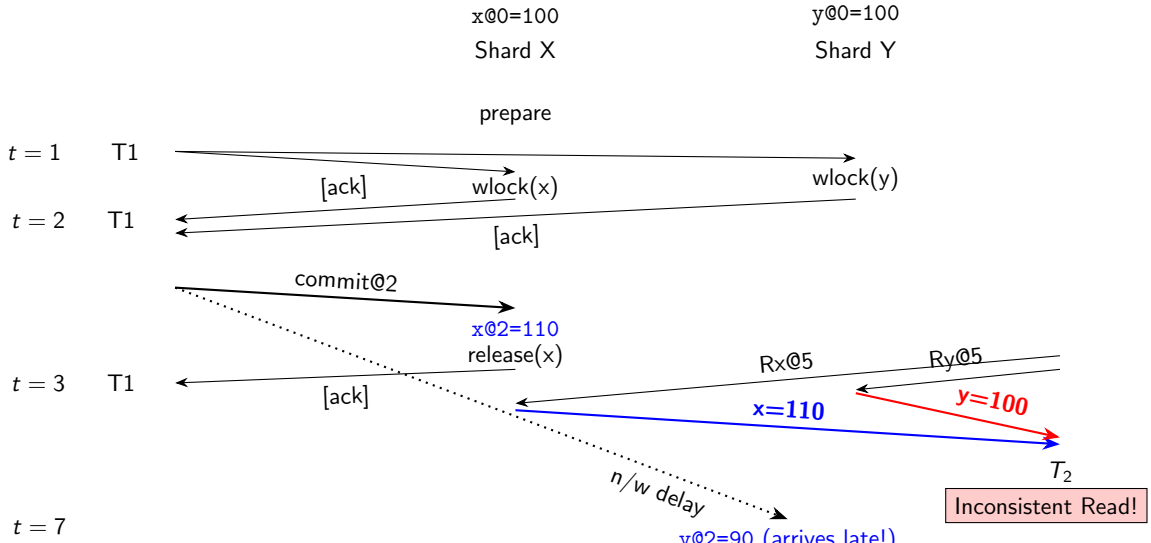
Step a: MVCC - The Ideal Case



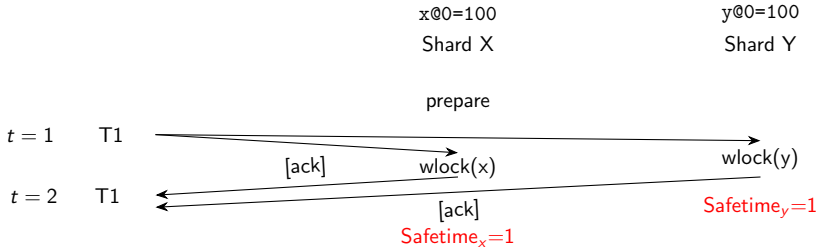
Step a: MVCC - The Ideal Case



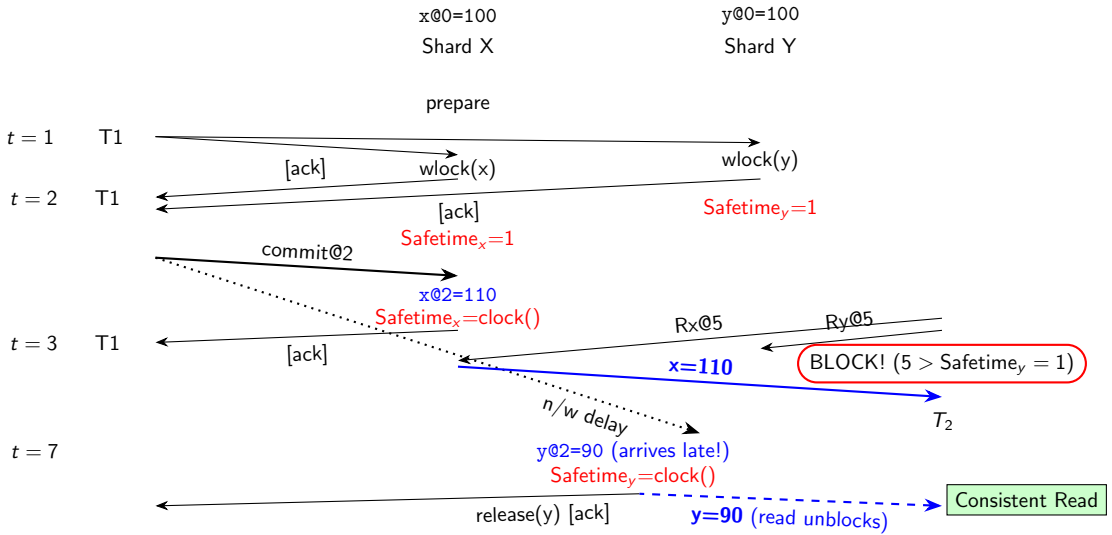
Step b: The 'Late Commit' Problem



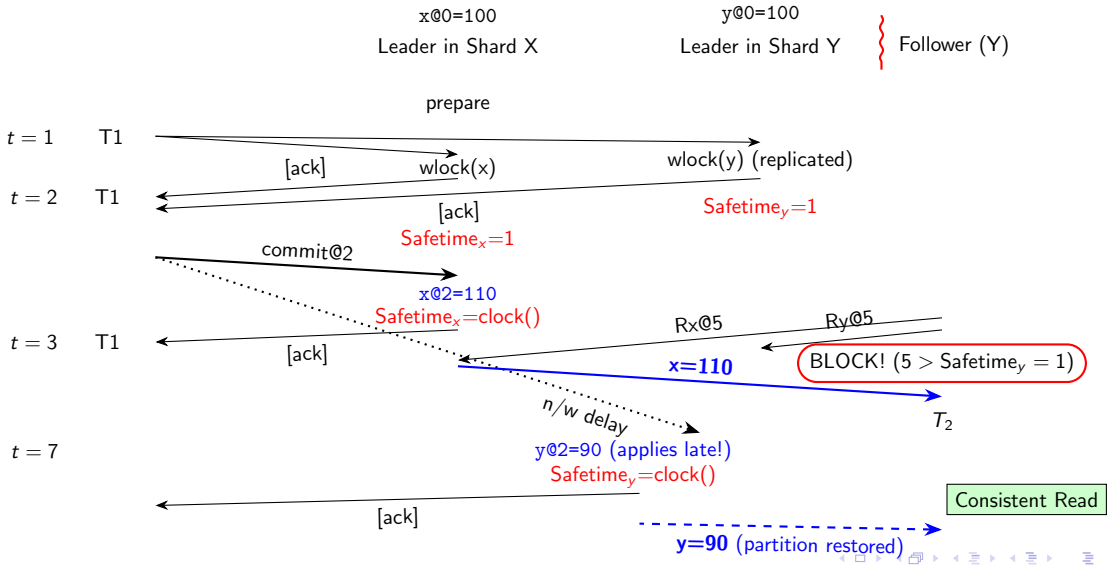
Step c: The Solution → SafeTime



Step c: The Solution → SafeTime



Step d: Leader (X) / Follower (Y) Read



External Consistency: The Proof

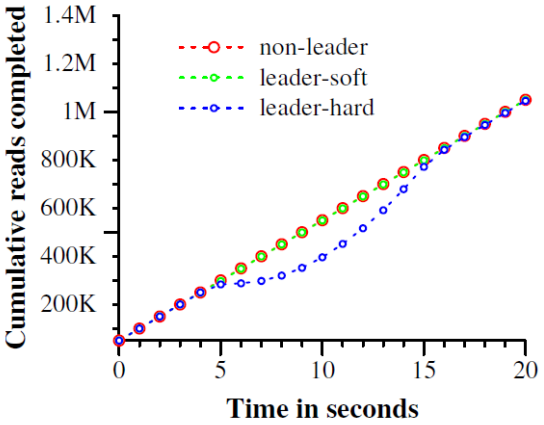
Let's prove: $t_{abs}(s_1^{commit}) < t_{abs}(s_2^{start}) \implies s_1 < s_2$

1. $s_1 < t_{abs}(s_1^{commit})$ by the Commit Wait rule on T_1.
2. $t_{abs}(s_1^{commit}) < t_{abs}(s_2^{start})$ by assumption.
3. $t_{abs}(s_2^{start}) \leq t_{abs}(s_2^{server})$ by causality.
4. $t_{abs}(s_2^{server}) \leq s_2$ by the Start Rule on T_2.

Conclusion (by transitivity)

$s_1 < t_{abs}(s_1^{commit}) < t_{abs}(s_2^{start}) \leq t_{abs}(s_2^{server}) \leq s_2 \implies s_1 < s_2$

Evaluation: Availability (Leader Failure)



Effect of Killing Servers

Experiment Setup

- 5 Zones ($Z_1..Z_5$), all leaders in Z_1.
- At $T = 5s$, a zone is killed.

non-leader Kill Z_2 (a non-leader zone). **No effect** on read throughput.

leader-soft Kill Z_1 with warning (leaders handoff first). **Minor dip** (3-4%).

leader-hard Kill Z_1 with no warning. **Throughput drops to 0**.

Evaluation: F1 Case Study

F1: Google's Advertising Backend

Spanner's first major customer was F1, a rewrite of Google's ad backend, previously on a manually-sharded MySQL database.

The Problem with MySQL

- The dataset (tens of TBs) was too large.
- Resharding was a **nightmare**: the last one took over 2 years of intense effort.
- Some data was moved to Bigtable, **compromising transactional behavior**.

Why F1 Chose Spanner

- **Removes manual resharding**.
- **Synchronous replication** and automatic failover.
- **Strong transactional semantics**, which F1 required.

Conclusion

- Spanner combines ideas from two communities:
  - **Database Community:** Semi-relational interface, transactions, SQL.
  - **Systems Community:** Scalability, automatic sharding, fault tolerance, consistent replication, wide-area distribution.
- It demonstrates that a globally-distributed database *can* provide strong, externally-consistent transactional guarantees.

Thank you.