Google Cloud

Spanner

A horizontally scalable, highly-available SQL Database

Ben Vandiver

Google

3 April 2025

What is Spanner?

Spanner is a SQL Database

```
SELECT COUNT(*) AS Count
FROM Artists AS a
WHERE LEFT(a.name,1) = ‘M’;
```

Spanner is Transactional & Consistent

1 Transaction A

2 Transaction D

3 Transaction B

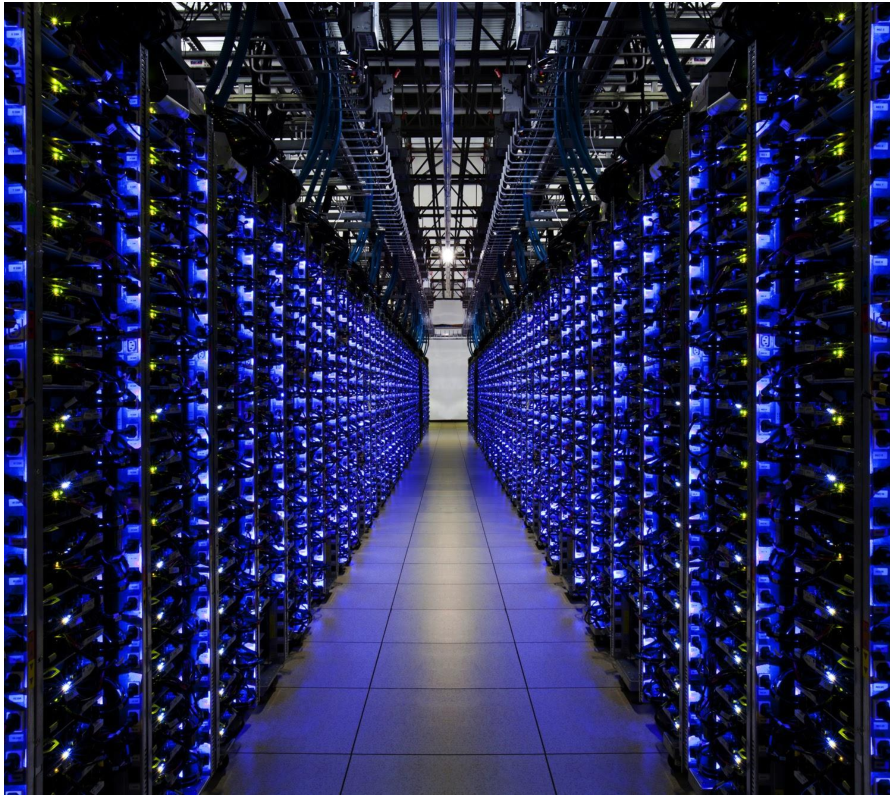
4 Transaction E

External Consistency
(stronger than linearizability and serializability)

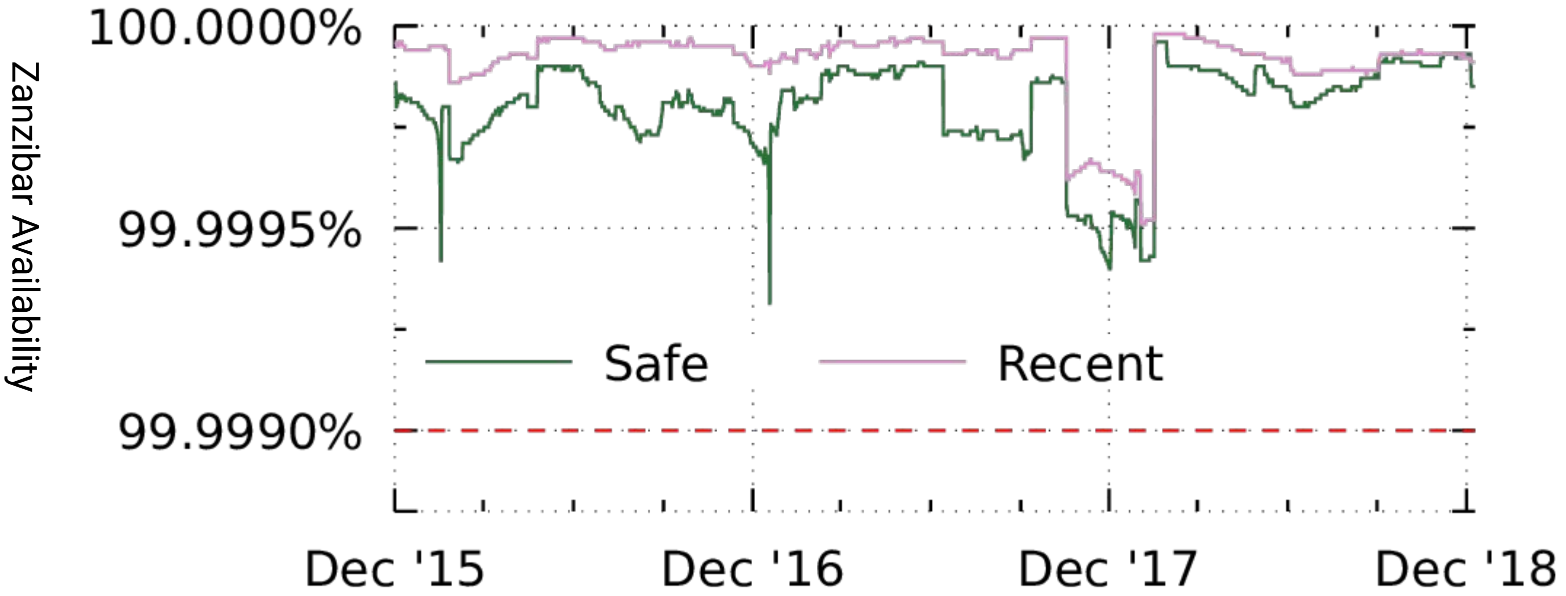
https://cloud.google.com/spanner/docs/true-time-external-consistency

Spanner is Geographically Replicated

Spanner is Horizontally Scalable



Zanzibar Availability on Spanner



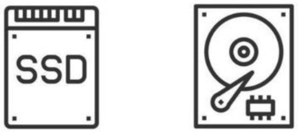
Availability over 3 years remained above 99.999% (“five 9’s”)

99.999% is ~5 minutes of downtime per year.

Pang, Cáceres, et al: [Zanzibar: Google's Consistent, Global Authorization System](), 2019 USENIX Annual Technical Conference

How Big is Spanner?

Spanner’s Scale



15 Exabytes

5 B QPS

What Makes Spanner so Big?

Photos Play Search Maps Ads YouTube Gmail Calendar Drive Docs Meet

What Makes Spanner so Big?

Photos Play Search Maps Ads YouTube Gmail Calendar Drive Docs Meet

Google Cloud

Cloud Control Plane

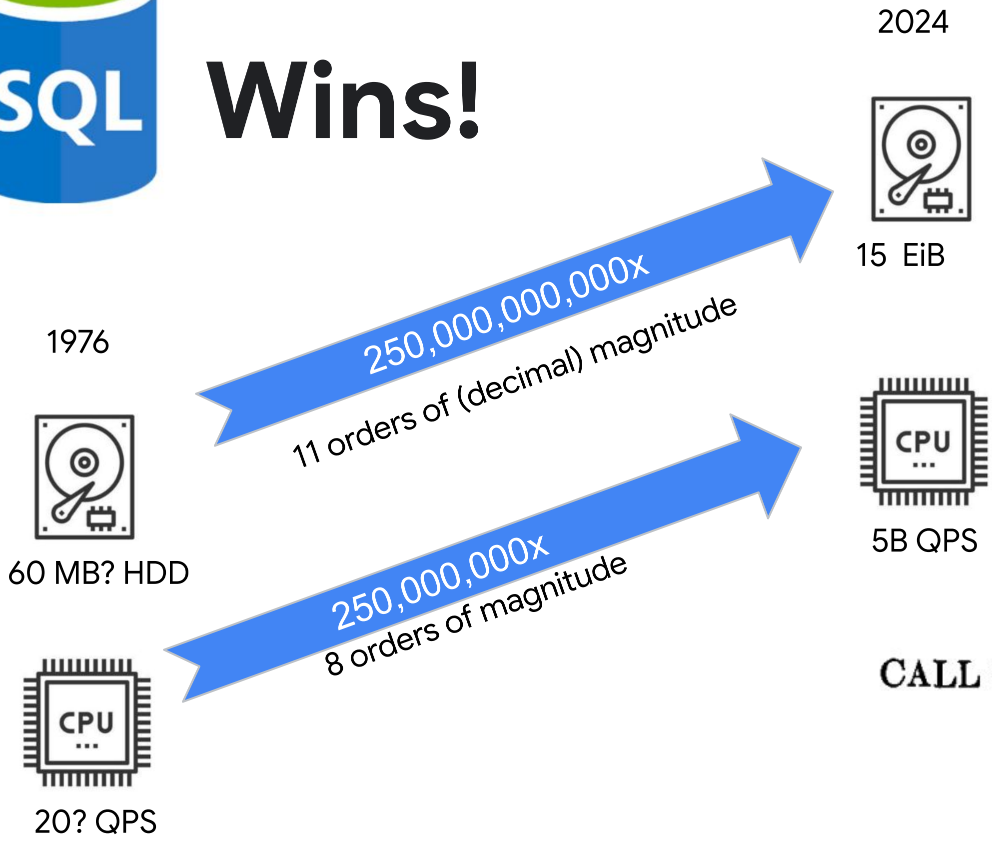
Cloud Spanner

Zanzibar

google3

Google's internal infrastructure

SQL Wins!




```
CALL SEQUEL('UNDERPAID(NAME, SAL) ←
        SELECT   NAME, SAL
        FROM     EMP
        WHERE    JOB = ''PROGRAMMER''
        AND      SAL < 10,000');
```

Astrahan et al: "System R: Relational Approach to Database Management" *ACM TODS 1*, 2 (June 1976)

Programmers Not So Much

```
CALL SEQUEL('UNDERPAID(NAME, SAL) ←
        SELECT   NAME, SAL
        FROM     EMP
        WHERE    JOB = ''PROGRAMMER''
        AND      SAL < 10,000');
```

1976

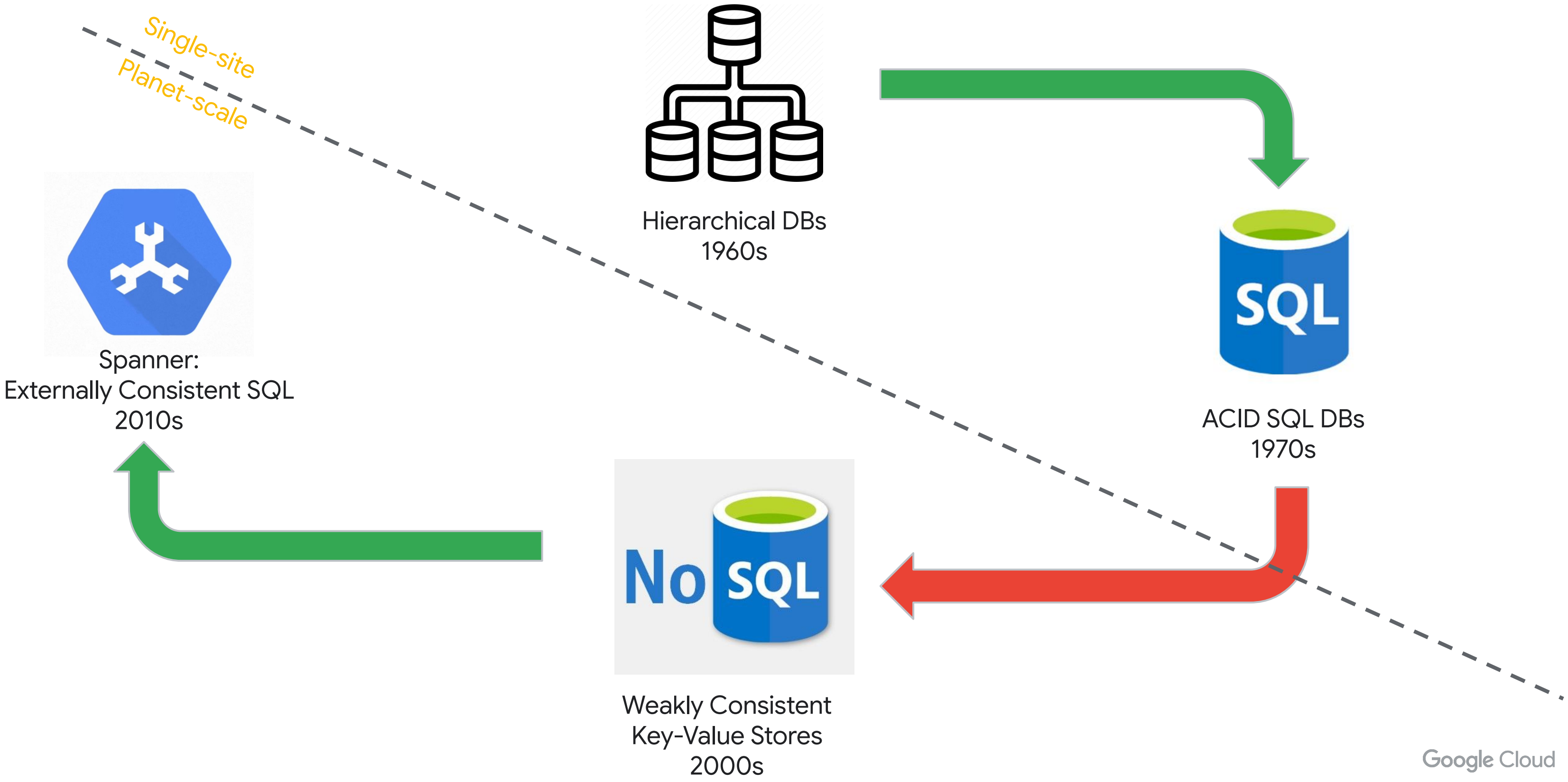
< $10,000

1 order of magnitude

2024

< $100,000

Coming Full Cycle



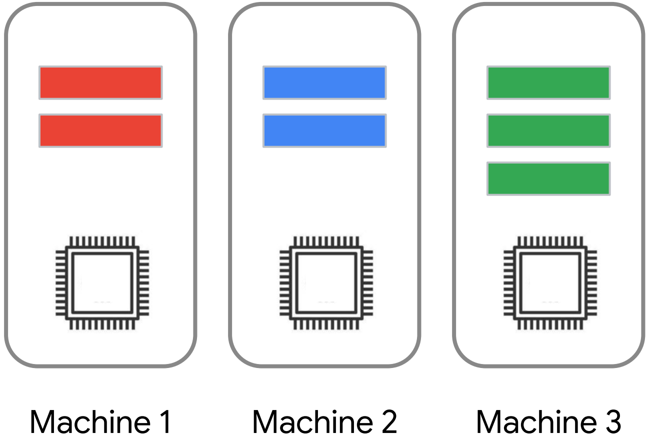
How Does Spanner Scale?

SQL Example (not so unrealistic)

```
CREATE TABLE Songs (
  ArtistId      INT64 NOT NULL,
  AlbumId       INT64 NOT NULL,
  TrackId       INT64 NOT NULL,
  SongName      STRING(MAX),
  ...
) PRIMARY KEY (
  ArtistId, AlbumId, TrackId)
```

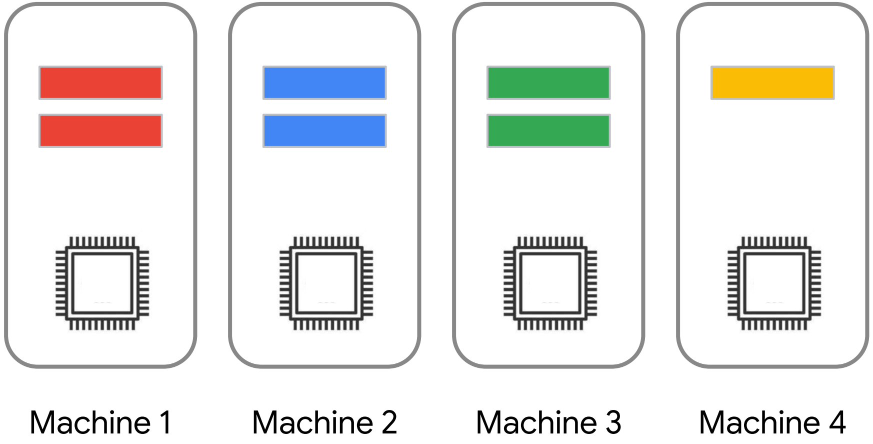
ArtistId	AlbumId	TrackId	SongName
1	1	1	A Night in Tunisia
1	2	1	Exuberante
3	1	1	Straight, No Chaser
3	2	1	All Blues
3	2	2	Flamenco Sketches
3	2	3	So What
5	1	1	Gangnam Style

Sharding



ArtistId	AlbumId	TrackId	SongName
1	1	1	A Night in Tunisia
1	2	1	Exuberante
3	1	1	Straight, No Chaser
3	2	1	All Blues
3	2	2	Flamenco Sketches
3	2	3	So What
5	1	1	Gangnam Style

Splitting (and Merging)



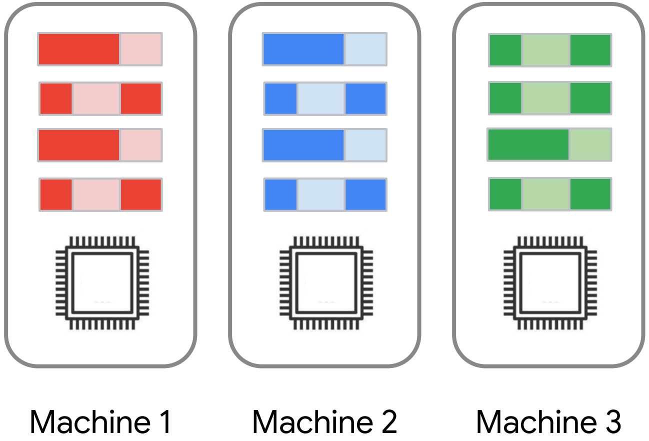
ArtistId	AlbumId	TrackId	SongName
1	1	1	A Night in Tunisia
1	2	1	Exuberante
3	1	1	Straight, No Chaser
3	2	1	All Blues
3	2	2	Flamenco Sketches
3	2	3	So What
5	1	1	Gangnam Style

Interleaving

```
CREATE TABLE Albums (
  ArtistId      INT64 NOT NULL,
  AlbumId       INT64 NOT NULL,
  AlbumName     STRING(MAX),
  ...
) PRIMARY KEY (
  ArtistId, AlbumId);
```

```
CREATE TABLE Songs (
  ArtistId      INT64 NOT NULL,
  AlbumId       INT64 NOT NULL,
  TrackId       INT64 NOT NULL,
  SongName      STRING(MAX),
  ...
) PRIMARY KEY (
  ArtistId, AlbumId, TrackId),
INTERLEAVE IN PARENT Albums;
```

Interleaving



ArtistId	AlbumId	AlbumName	TrackId	SongName
1	1	Best of Dizzy Gillespie		
1	1		1	A Night in Tunisia
1	2	Afro-Cuban Moods		
1	2		1	Exuberante
3	1	Milestones		
3	1		1	Straight, No Chaser
3	2	All Blues		
3	2		1	All Blues
3	2		2	Flamenco Sketches
3	2		3	So What
5	1	Psy 6 (Six Rules)		
5	1		1	Gangnam Style

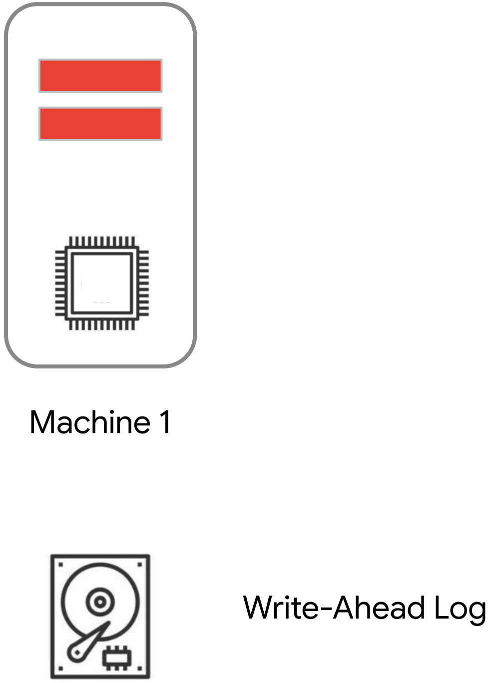
Transactions &
High Availability

Transactions: Single-Shard

Well-understood:

- Transaction Manager with pessimistic locking
- Strict 2-Phase Locking
- Write-Ahead Logging to persistent storage (disk)

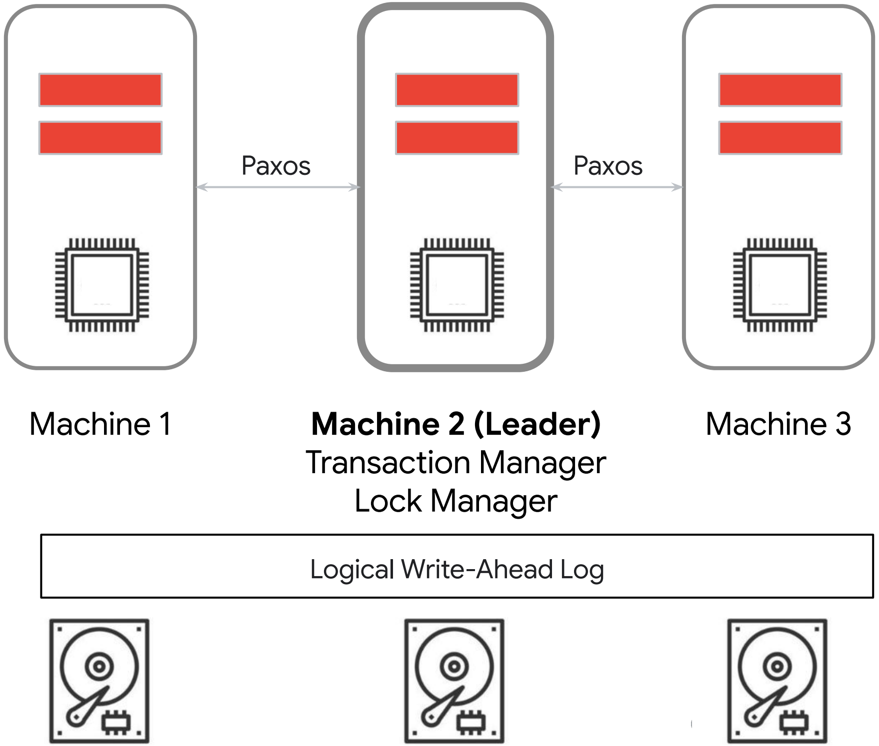
But what about machine failures? Want copies of the data on multiple machines (ideally in different data centers).



Transactions: Paxos

Answer: Paxos (or Raft)

- Synchronous replication protocol
- Gracefully deals with failures
  - Makes progress as long as majority of replicas are alive
- Elects a leader to manage writes
  - Transaction Manager and Lock Manager run here



Paxos in a Nutshell

Manages a shared log of writes. Every replica applies log entries in order.

Consists of N = 2f + 1 nodes

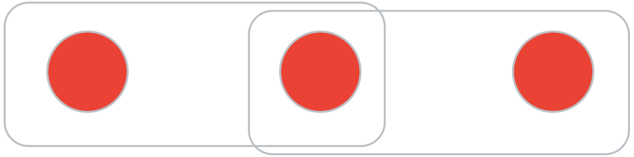
- Can tolerate f failures

Examples:

- N=3 nodes: tolerate 1 failed node
- N=5 nodes: tolerate 2 failed nodes

Quorum: Majority of nodes (f + 1)

Observation: Intersection between any two quorums is at least one node.



Distributed Transactions

What about transactions that span multiple shards?

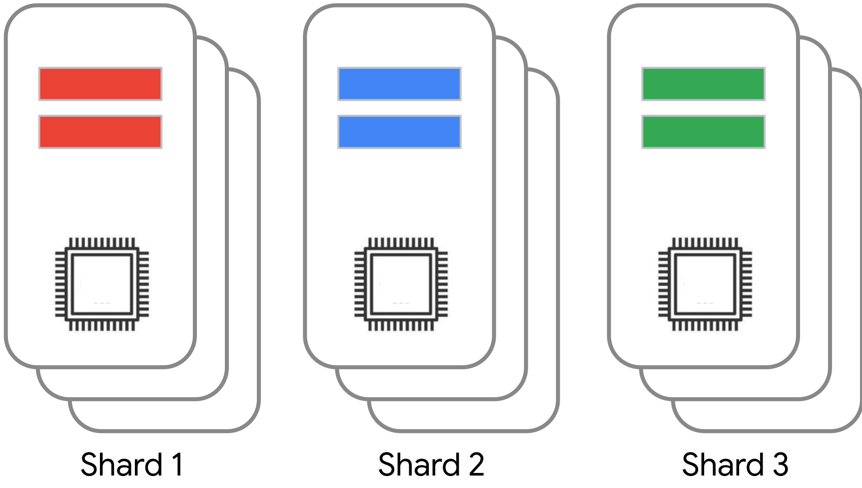
Answer: 2-Phase Commit (2PC)

- Participants are paxos groups (not individual machines)
- One participant picked as Coordinator
- All participants (durably) prepare and notify coordinator
    - Promise not to release locks
- Coordinator commits transaction once all participants have prepared, then notifies participants

Biggest down-side of 2PC:

- Gets stuck if coordinator becomes unavailable

We just made every participants (and coordinator) highly available through Paxos!



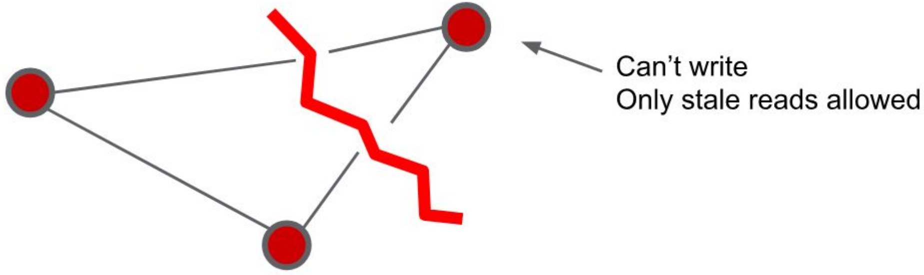
CAP Theorem

C - Consistent
- All nodes have identical data, strong reads

A - Available
- Every request gets a response

P - Partition Tolerant
- Continue to function (correctly) even if a network partition occurs.

Pick 2!



Spanner is a **CP** system.

SQL interface

Common SQL dialect

- Standards-compliant
- Type system aligned with programming languages
  - INT64, FLOAT, STRING (UTF8), TIMESTAMP (nanoseconds)
  - Reduces impedance mismatch
- First-class support for nested data
  - ARRAY and STRUCT types
  - Protocol Buffers: schematized binary objects

    Significant language design work across teams
- Shared with other Google systems: BigQuery, F1, etc.

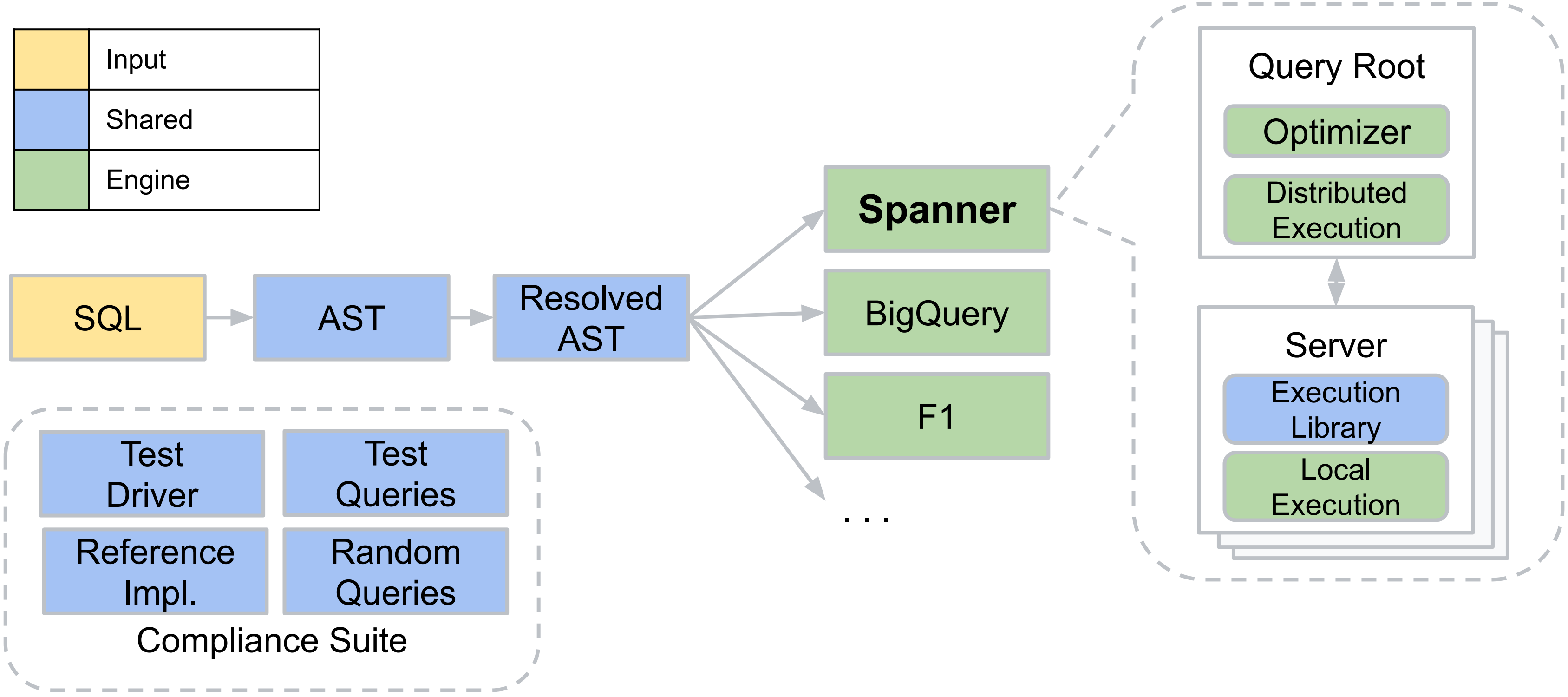
Sample query: name & titles

```
SELECT s.SingerName,
       ARRAY(SELECT a.AlbumTitle
             FROM Albums a
             WHERE a.SingerId = s.SingerId) titles
FROM Singers s
WHERE s.SingerId BETWEEN 1 AND 5
```

SingerName STRING	**titles** ARRAY<STRING>
Beatles	[Help!, Abbey Road]
U2	[ ]
Pink Floyd	[The Wall]

- Easier to use than outer joins or multiple roundtrips

Same query semantics across systems



Google

Distributed Execution

Distributed query execution

- Tightly coupled architecture
  - Query processor inside the database server
  - Typical design for standalone DBMSes (vs. distributed systems)
- Challenge of scale: data never sits still
  - Continuous resharding (due to load, capacity, config changes, ...)
  - Shard boundaries may change while query is running
  - Shards may become temporarily unavailable during query execution
  - Alternative replicas: near/far, loaded/idle, caught-up/behind
- Mechanisms used in Spanner
  - Query routing: key-range rpcs + range extraction
  - Parallelizing execution: partition work by shards, push it down
  - Dealing with failures: restartable query processing

Query routing: key-range rpcs

- Routes requests to row ranges
  - E.g., `WHERE SingerId BETWEEN @low AND @high`
- Hides complexity of locating data
- Finds nearest, sufficiently up-to-date replica for given concurrency mode
- Retries automatically
  - Unavailability, data movement, schema updates, ...
- Clients cache sharding information
- Clients cache "location hints" for queries
  - Send query to right server without extra hops or query analysis
  - E.g., `Singers/SingerId[@low]`

Query routing: range extraction

```
SELECT * FROM Albums
WHERE (SingerId = 1 AND AlbumId >= 10) OR
      (SingerId IN (2,3) AND AlbumId != 0)
```

- Also used for restricting scan ranges
- Computed at runtime
  - May access data
- Uses efficient data structure
  - Filter tree (in the paper)

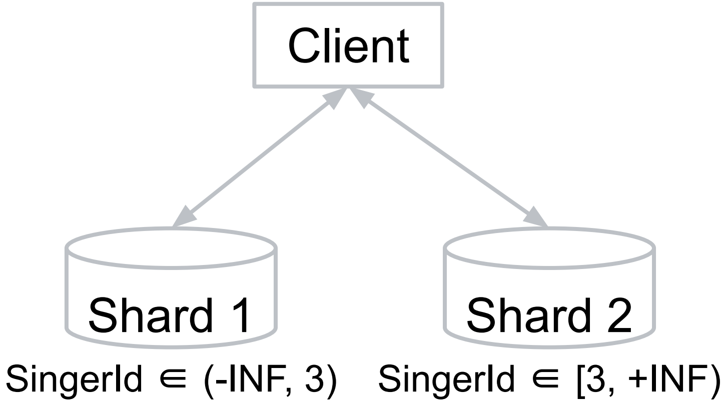
SingerId	AlbumId
[1..1]	[10, +INF)
[2..2]	(-INF, 0)
[2..2]	(0, +INF)
[3..3]	(-INF, 0)
[3..3]	(0, +INF)

Google

Parallelizing Execution

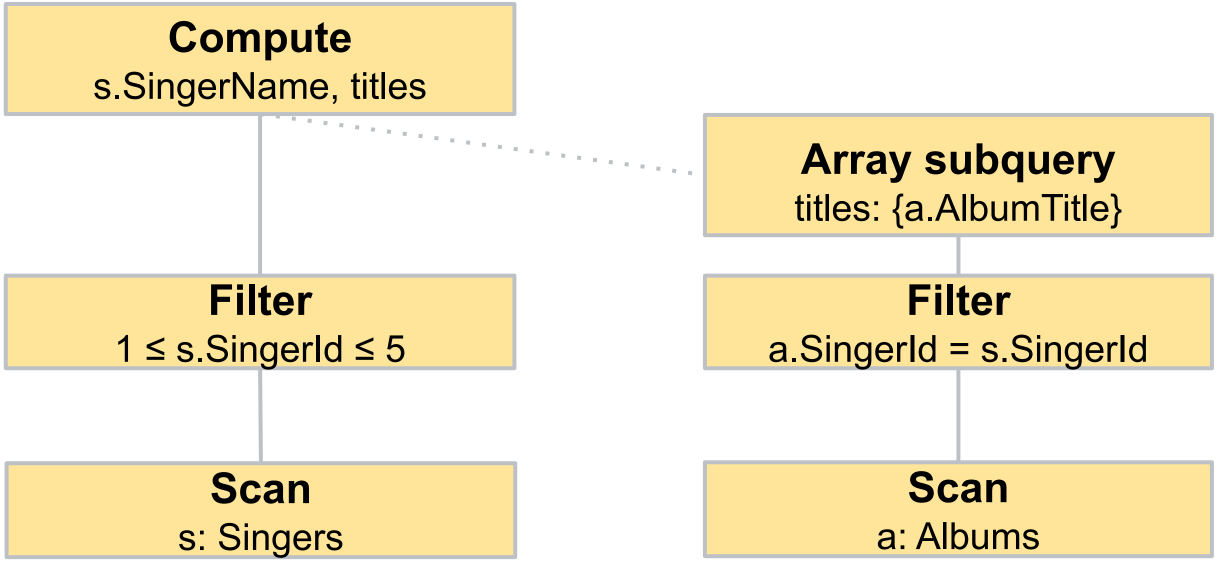
Parallelizing execution

```
SELECT SingerName, ARRAY(SELECT ...) titles
FROM Singers WHERE SingerId BETWEEN 1 AND 5
```

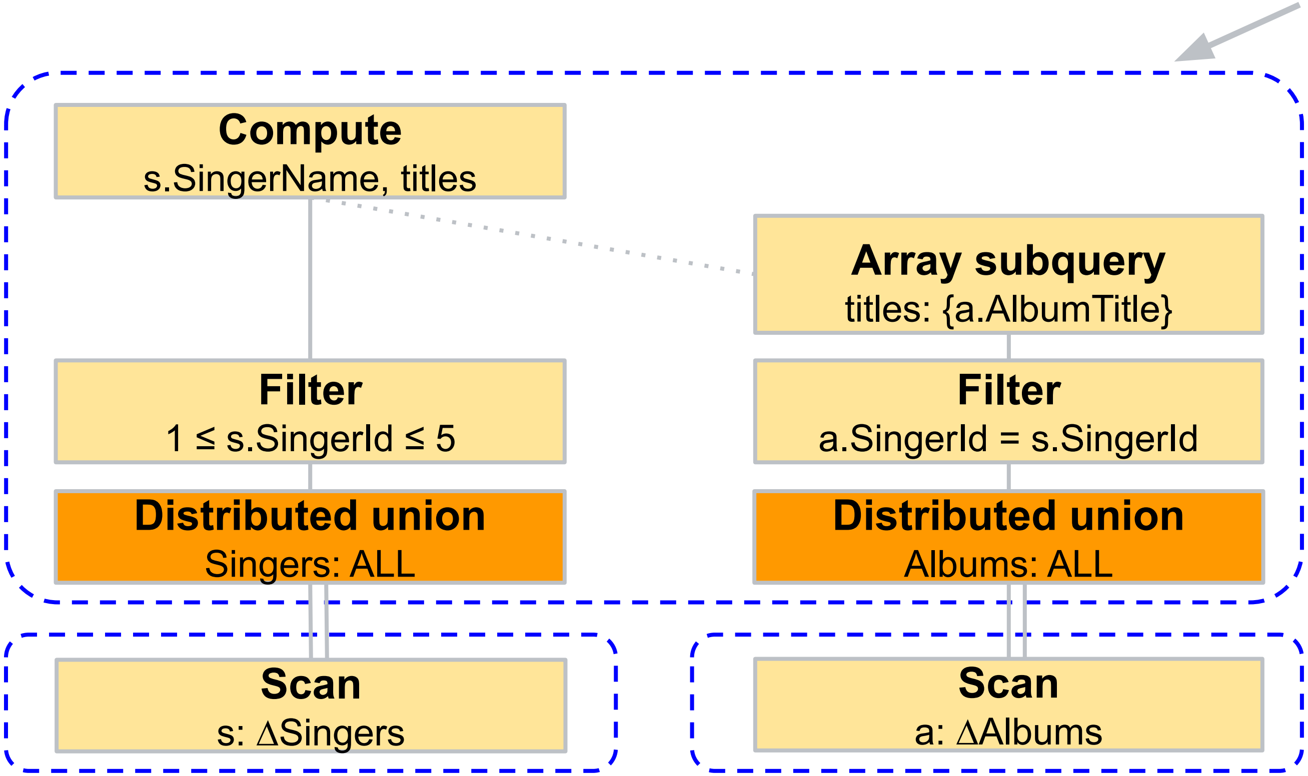


- Assume fixed shard boundaries for now

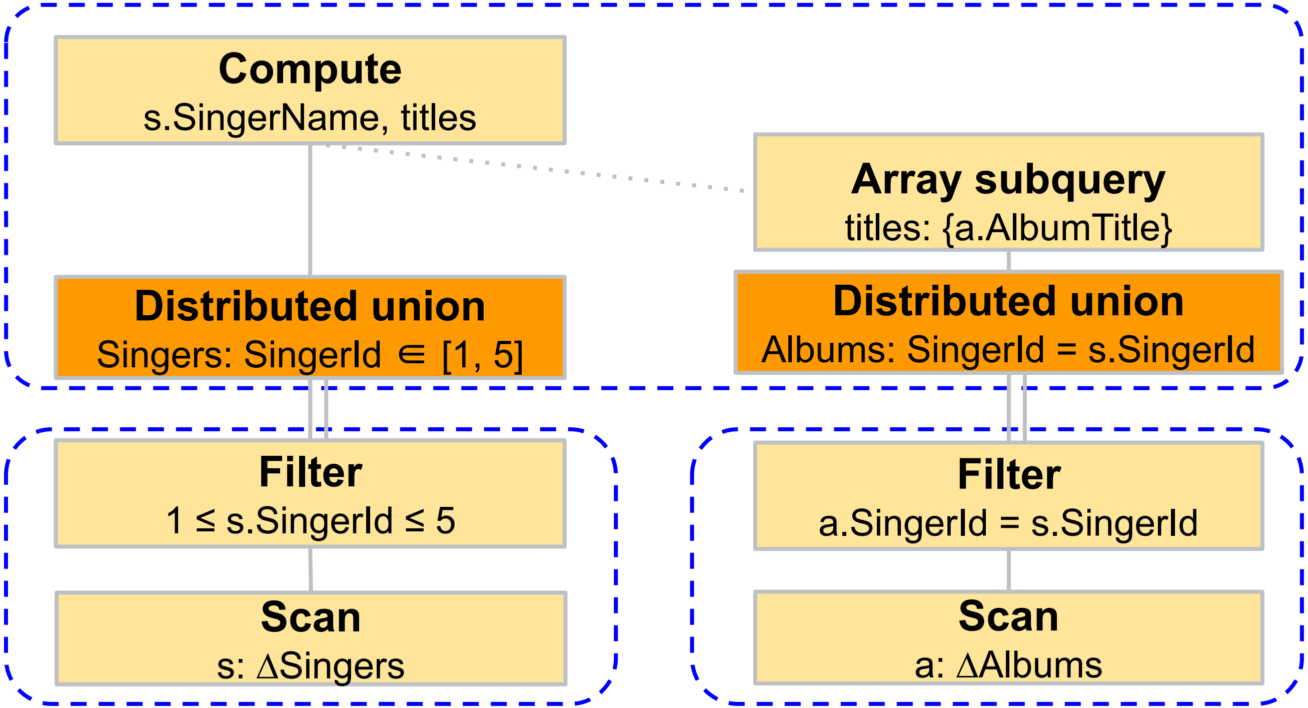
Initial logical plan



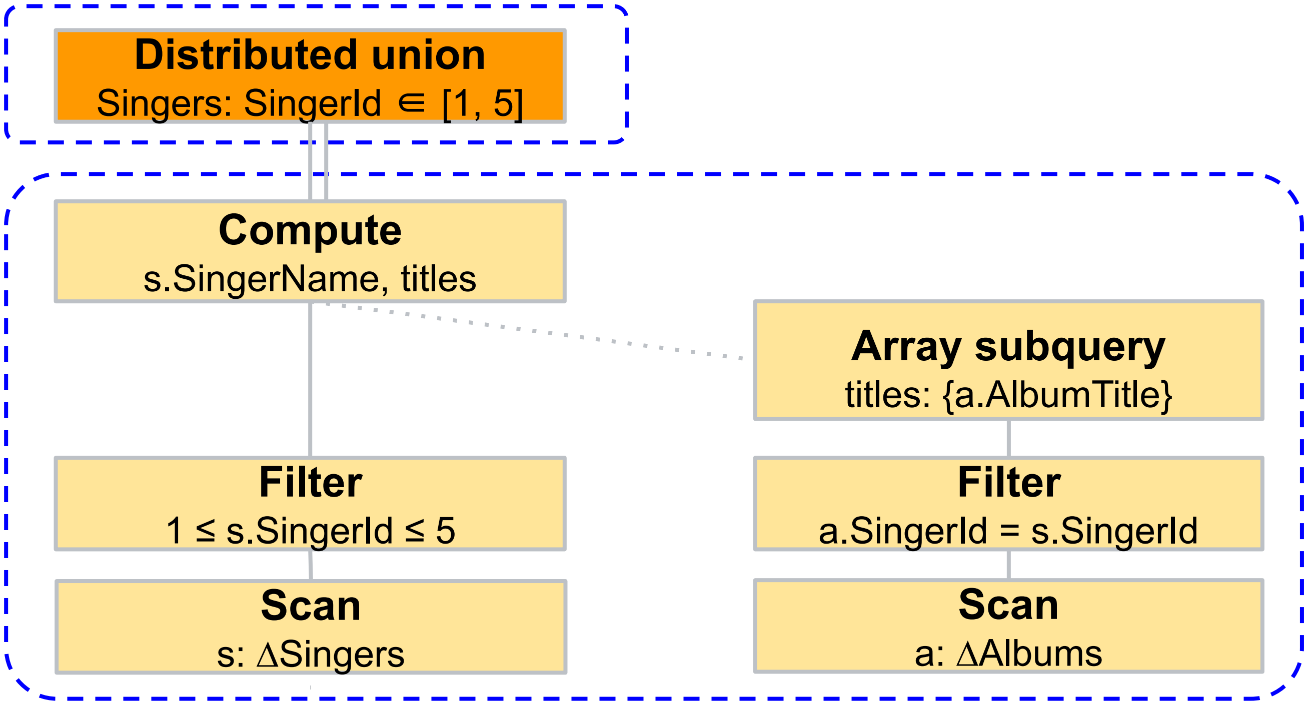
Distributed union operator



Push work to shards, extract distribution ranges



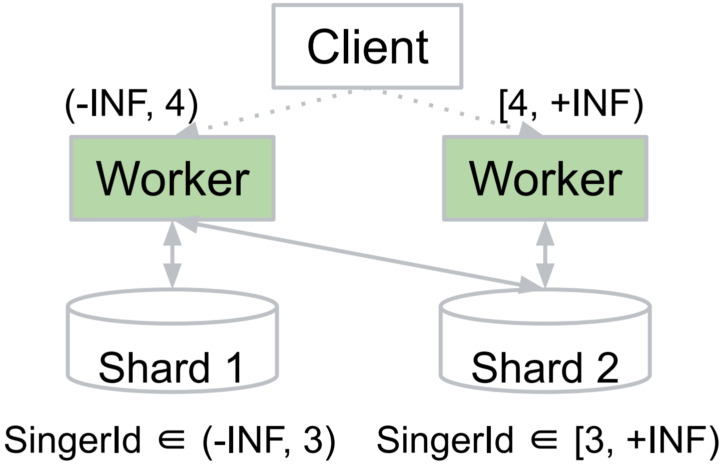
Exploiting co-location



Parallel-consumer API

```
SELECT SingerName, ARRAY(SELECT ...) titles
FROM Singers WHERE SingerId BETWEEN 1 AND 5
```

- Root-partitionable query:
  Q(Union of ΔT) = Union of Q(ΔT)
- Same result up to order of rows
- Another main distribution operator: Distributed Cross Apply



Restartable snapshot queries

Query restarts: overview

- Automatic compensation for failures
- For snapshot queries only
- Server yields "restart token" with each result batch
- Client resumes query execution after consuming partial results
- Contract: omit previously returned rows
  - No repeatability guarantee for subsequent rows

SingerName STRING	**titles** ARRAY<STRING>
Beatles	[Help!, Abbey Road]
U2	[ ]
Pink Floyd	[The Wall]

restart

Query restarts: implementation challenges

- Naive solutions don't work well for "large" queries
  - Buffer final result, persist intermediate results, count rows, etc.
- Instead: efficiently capture distributed state of query execution
- Dynamic resharding
  - May restart on different row range
- Non-determinism
  - Memory size, parallelism, computer architecture, numerics, ...
- Restarts across server versions
  - Query plans, execution algorithms

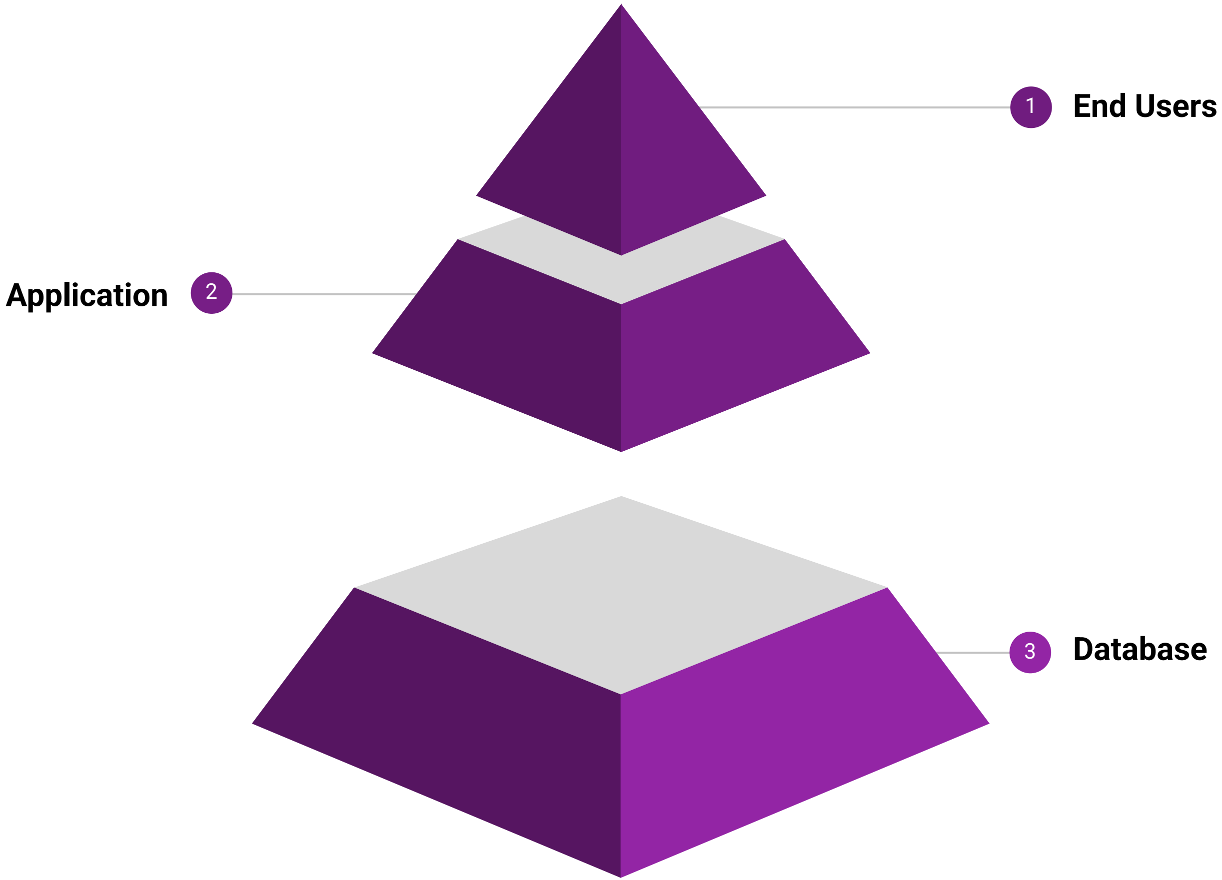
Query restarts: hard but worth it

- Hide transient failures
- No retry loops: simpler programming model
- Streaming pagination
- Ensure forward progress for important class of long-running queries

- Improve tail latency of online requests
- Low-impact rolling server upgrades

Building Database Systems

Location



Database System Contract

Goal: Absorb complexity from application designers

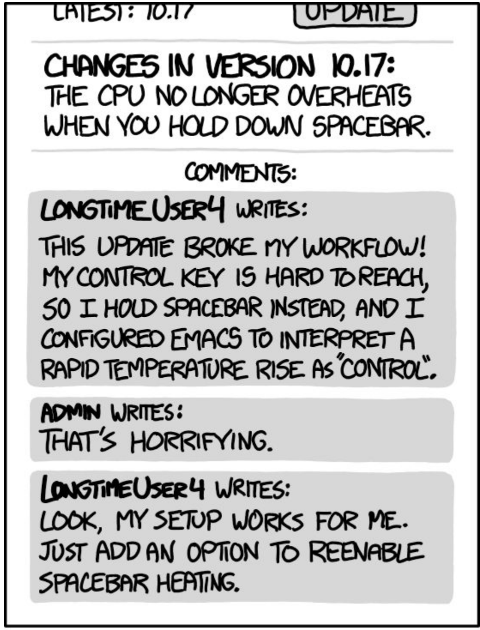
- Failures - Replication, Transactions
- Performance - Efficient data layout, Query Optimization
- Scalability - Sharding, Routing
- Security & Compliance - Regulatory, Fine-grained ACLs

What are we building?

- Operating System + Compiler + Distributed System
- Interface surface area is immense
- SQL is a standard - OK, not really.

Hyrum's Law

With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

[HTTPS://XKCD.COM/1172/](https://xkcd.com/1172/)

Complexity

- Respect Complexity
  - No one person fully understands Spanner. Not even close.
- Spend "complexity budget" where it matters
- Databases are heavily interconnected systems
  - Hard to tightly componentize - "leaky abstractions"
  - Everything breaks Backup/Restore

Tests, Tests, Tests

- Randomized Testing
  - **Random Query, Data, and Schema generators**

    Generate syntactically and semantically valid queries and data

    (eg queries actually return rows)
  - **Inject Faults**

    Anything that can fail or change should
  - **Validate with simpler emulator**

    Check the answers against a much simpler system with the same semantics

For more: "Randomized Testing of Cloud Spanner" Jay Corbett (Medium post)

Tests, Tests, Tests

- Integrity Checks
  - **Validate structures against each other internally**

    Index vs table, generated columns vs expressions, check constraints, foreign keys, etc.
  - **Use disaggregated compute to avoid workload impact**
- Log like crazy
  - It may take a while to notice an issue

Plan Stability: Avoiding Regressions

- *"We have millions of query plans in production,*

  how do we update the optimizer safely?"
- Pin Plans? Pin Optimizer version? Pin feature flags?
- Disaggregation helps:
  - pin plans
  - async validate nothing regresses
  - unpin as you validate

Questions?