F1

A Distributed SQL Database That Scales

RedBlue: Latency Over Consistency

Problem: online services distribute and replicate state across geographically diverse data centers, direct user requests to the optimal site

- This ensures low latency, but doesn’t ensure consistency
- There exists an inherent tension between consistency and latency
- Potential solution: allow multiple levels of consistency to exist
  - Some operations can be executed optimistically
  - Others always require cross-site synchronization

RedBlue: Solution

Part 1: RedBlue consistency

- Blue operations are fast (and eventually consistent)
- Red operations are strongly consistent (and slow)

Part 2: Use fast operations whenever possible, resort to strong consistency only when needed

- Identify conditions that suggest when operations can be blue and must be red

Part 3: Method that increases the space of potential blue operations by breaking them into a separate generator and shadow phases.

RedBlue: Solution

Part 3 (continued): Operations are decomposed into two components.

1.) Generator operation - identifies the changes the original operation should make, but has no side effects itself
2.) Shadow operation - performs the identified changes and is replicated to all sites. These operations are the only ones that are colored red/blue.

This system allows for fine-grained classification of operations.

PNUTS

- Parallel and geographically distributed database system for Yahoo’s web applications
- Data storage: organized as hashed/ordered tables
- Low latency even for a high number of concurrent requests
- Consistency: per-record guarantees
- Centrally managed, geographically distributed

F1 & Spanner: Related Work

- F1’s relational query execution techniques are similar to those in shared-nothing literature with some key differences such as ignoring of interesting order and the absence of copartitioned data
- F1’s NoSQL capabilities are similar to those of other scalable key-value stores
- Hierarchical schema and clustering properties are similar to Megastore

F1 & Spanner: Related Work (continued)

BigTable: distributed storage system for managing structured that that is designed to scale to a very large size (petabytes of data cross thousands of commodity servers).

Can be difficult to use for certain kinds of applications:

- Applications with complex, evolving schemas
- Applications that want strong consistency in the wide-area replication

HBase is another non-relational distributed database that is modeled after BigTable and is open sourced.

F1: Related Work (continued)

Megastore: storage system that combines the scalability of NoSQL datastores with the convenience of traditional relational database management systems

- Designed to meet the requirements of modern interactive online services:
  - Applications must be highly scalable
  - Must compete for users (rapid development is key)
  - Must be responsive (low latencies)
  - Must be consistent (for the sake of user-facing applications
- Provides strong consistency guarantees and is highly available
- However: relatively poor write throughput as a result

F1: Related Work (continued)

- **Geo-distributed analytics:** longer response latencies of the order of minutes are acceptable, whereas in F1 it should be as quick as possible (ideally no more than a few seconds)
- In F1, transactions are viewed as necessary (because eventually consistent databases cause programmer errors), whereas other data stores prioritize lower latencies to consistency (e.g. - **Eiger** enables causal consistency, which is weaker than F1, but leads to lower latency)
- **Pileus:** a system that automatically adjusts the consistency level to achieve the desired latency goal expressed as an SLA.

Spanner: Google’s Globally Distributed Database

- Database that shards data across many sets of Paxos state machines in datacenters across the world
- Uses replication for:
  - Global availability
  - Geographic locality
  - Fault tolerance
- Dynamically reshards data as the number of servers change
- Automatically migrates data across machines to balance load (and for fault tolerance)

Spanner: Key Features

- Replication configurations for data can be dynamically controlled at a fine-grain
  - Applications can specify constraints such as which datacenters control which data, etc.
- Provides externally consistent reads and writes
- Provides globally consistent reads and writes and globally consistent reads. Allows Spanner to support:
  - Consistent backups
  - Consistent MapReduce executions
  - Atomic schema updates

The Problem

- Want to avoid performance degradation when using databases that are scattered across the world
- Existing geo-distributed databases with sharded MySQL implementations don’t scale well

F1

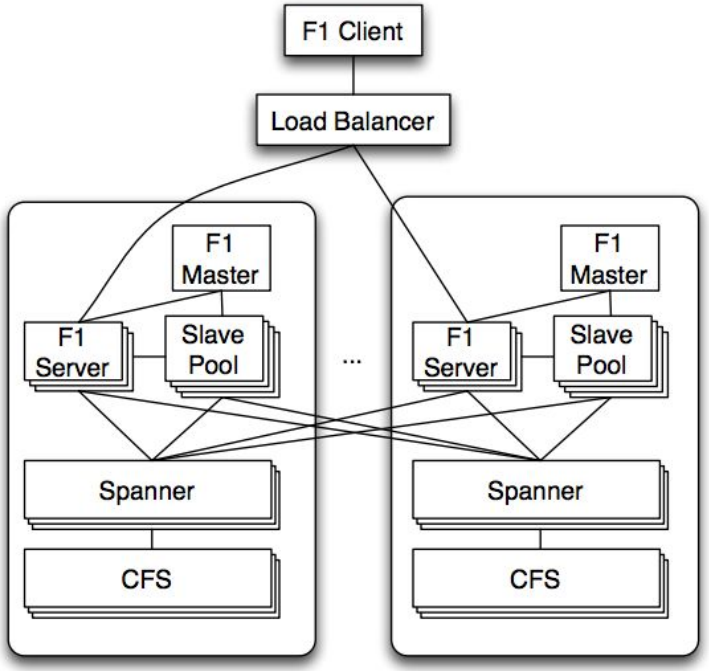
- Fault-tolerant globally-distributed OLTP and OLAP database built at Google, on top of Spanner
- Used as the new storage system for Google’s AdWords
- Replaces the sharded MySQL implementation that doesn’t meet new requirements (for scalability and performance)

F1: Design Goals

1. **Scalability:** system should scale up by just adding resources
2. **Availability:** system must never go down for any reason
3. **Consistency:** system must provide ACID transactions
4. **Usability:** system must provide full SQL query support

Prior to this work, it was believed that these goals were mutually exclusive.

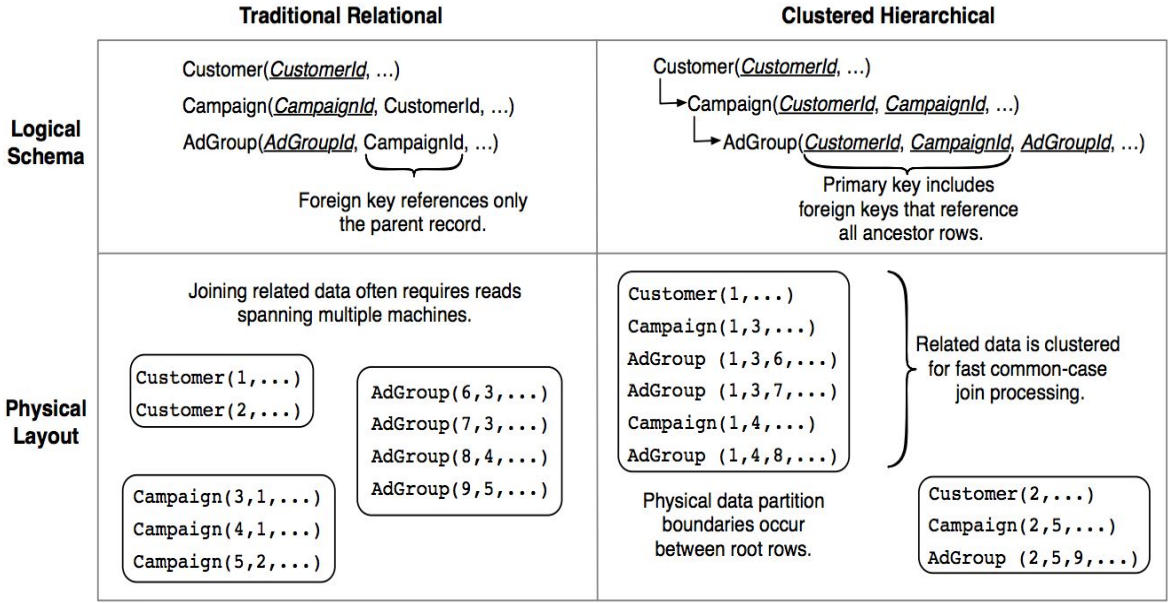
F1: Architecture



- Users interact with F1 through its client library, sends requests to servers
- F1 servers connected to nearby datacenters whenever possible
- F1 servers are typically co-located in the same set of datacenters are the Spanner servers storing the data
- Spanner servers in each datacenter retrieve their data from the Colossus File System (CFS)
- F1 servers are mostly stateless, so clients can communicate with different servers for each request
- F1 servers can be quickly added or removed in response to the total load, because they don’t own data

F1: Data Model

- **Logically:** tables are organized hierarchically. Can't be arbitrarily interleaved. Child table must have a foreign key to parent as prefix of its primary key.
- **Physically:** each child table is clustered with and interleaved within rows of parent table



F1: Schema Changes

- All schema changes are fully non-blocking, and are applied asynchronously, on different F1 servers at different times
- To prevent anomalies, schema change algorithm:
  - Enforces that across all F1 servers, at most two different schemas are active (via granting leases to schema and ensuring that no server uses a schema after lease expiry)
  - Subdividing each schema change into multiple phases where consecutive pairs of phases are mutually compatible and cannot cause anomalies

F1: Transactions

F1 supports the following three transactions:

- **Snapshot transactions:** read-only transactions with snapshot semantics, reading repeatable data as of a fixed Spanner snapshot timestamp (default for SQL queries and MapReduces). Read-only.
- **Pessimistic transactions:** use a stateful communications protocol that requires holding locks, so all requests in a single pessimistic transaction get directed to the same F1 server. If the F1 server restarts, the pessimistic transaction aborts. Reads in pessimistic transactions can request either shared or exclusive locks. Only these types of transactions are “general” (can do arbitrary reads/writes).
- **Optimistic transactions:** consist of a read phase, which can take arbitrarily long and never takes Spanner locks, and then a short write phase. Only allow this single read followed by a single write.

Tradeoff: these transactions typically have better performance, but they come at the cost of a toned-done version of the API.

F1: Change History

- Change History is a first-class feature at the DB level
- All tables are change-tracked by default (although specific ones can be opted out)
- Every transaction creates 1+ ChangeBatch Protocol Buffers, which include the primary key & before + after values of changed columns for each updated row

F1: Query Execution Tradeoffs

- Centralized execution is meant for short-running OLTP queries
  - Can run the query on a single F1 node and get low latency
- Distributed execution is designed for long-running OLAP queries, and possible when F1 is used to run MapReduce style queries.
  - Take the hit of high latency with the benefit of increased parallelism and throughput

F1: Evaluation

- Users can expect read latencies of 5-10 ms, while commit latencies are expected to vary and are determined by network latencies between datacenters
- Overall user facing latency averages 200 ms which is similar to the preceding MySQL system (despite increase in database access latency)
- For non-interactive applications with bulky updates, optimize for throughput rather than latency
- Generalizability past AdWords use-case is unclear

F1: Remote Data Accesses

- Remote data accesses are affected by highly variable network latency
- Network latency is mitigated by using extensive amounts of batching
- Network based storage is distributed over many disks, so it’s less likely that multiple parallel data accesses contend for the same resources
    - In fact, they often result in a close to linear speedup
    - This speedup continues until the underlying system is actually overloaded

F1: Hierarchical Table Joins

- F1's hierarchical data model allows it to efficiently join a parent table with a descendant table
  - Uses their shared primary key prefix to accomplish this
  - Only takes a single request to Spanner, which in turn requests data from both tables
  - Spanner will return data to F1 in an interleaved order
- Cluster join: a merge-join-like algorithm that F1 uses to interpret the stream returned by Spanner
  - Buffers one row from each table and returns the joined results in a streaming fashion as the Spanner input data is received
- Any number of tables can be joined this way using just a single request as long as they fall on a single ancestry path in the table hierarchy

Next Steps

- Resource costs are higher in F1 since queries typically require an order of magnitude more CPU than similar MySQL queries.
  - MySQL stores data uncompressed on local disk (bottleneck = disk)
  - F1 starts with data compressed on disk and goes through several layers of decompressing, processing, recompressing, and sending over network (high CPU cost)
- Next step: to improve CPU efficiency of F1 to the point where it's comparable to MySQL
- Next step: better performance for high-update geo-distributed workloads
  - F1's improvements would only help in snapshot or optimistic transactions
  - Both optimistic and snapshot transactions are meant for read-mostly workloads