F1: A Distributed SQL Database That Scales

Jeff Shute, Chad Whipkey, David Menestrina, Radek Vingralek, Eric Rollins, Stephan Ellner, Traian Stancescu, Bart Samwel, Mircea Oancea, John Cieslewicz, Himani Apte, Ben Handy, Kyle Littlefield, Ian Rae*

Google, Inc.
*University of Wisconsin-Madison

ABSTRACT

F1 is a distributed relational database system built at Google to support the AdWords business. F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and usability of traditional SQL databases. F1 is built on Spanner, which provides synchronous cross-datacenter replication and strong consistency. Synchronous replication implies higher commit latency, but we mitigate that latency by using a hierarchical schema model with structured data types and through smart application design. F1 also includes a fully functional distributed SQL query engine and automatic change tracking and publishing.

1. INTRODUCTION

F1[1] is a fault-tolerant globally-distributed OLTP and OLAP database built at Google as the new storage system for Google's AdWords system. It was designed to replace a sharded MySQL implementation that was not able to meet our growing scalability and reliability requirements.

The key goals of F1's design are:

1. **Scalability**: The system must be able to scale up, trivially and transparently, just by adding resources. Our sharded database based on MySQL was hard to scale up, and even more difficult to rebalance. Our users needed complex queries and joins, which meant they had to carefully shard their data, and resharding data without breaking applications was challenging.
2. **Availability**: The system must never go down for any reason – datacenter outages, planned maintenance, schema changes, etc. The system stores data for Google's core business. Any downtime has a significant revenue impact.
3. **Consistency**: The system must provide ACID transactions, and must always present applications with consistent and correct data.
   Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.
4. **Usability**: The system must provide full SQL query support and other functionality users expect from a SQL database. Features like indexes and ad hoc query are not just nice to have, but absolute requirements for our business.

Recent publications have suggested that these design goals are mutually exclusive [5, 11, 23]. A key contribution of this paper is to show how we achieved all of these goals in F1's design, and where we made trade-offs and sacrifices. The name F1 comes from genetics, where a *Filial 1 hybrid* is the first generation offspring resulting from a cross mating of distinctly different parental types. The F1 database system is indeed such a hybrid, combining the best aspects of traditional relational databases and scalable NoSQL systems like Bigtable [6].

F1 is built on top of Spanner [7], which provides extremely scalable data storage, synchronous replication, and strong consistency and ordering properties. F1 inherits those features from Spanner and adds several more:

- Distributed SQL queries, including joining data from external data sources
- Transactionally consistent secondary indexes
- Asynchronous schema changes including database reorganizations
- Optimistic transactions
- Automatic change history recording and publishing

Our design choices in F1 result in higher latency for typical reads and writes. We have developed techniques to hide that increased latency, and we found that user-facing transactions can be made to perform as well as in our previous MySQL system:

- An F1 schema makes data clustering explicit, using tables with hierarchical relationships and columns with structured data types. This clustering improves data locality and reduces the number and cost of RPCs required to read remote data.

[1] Previously described briefly in [22].

Proceedings of the VLDB Endowment, Vol. 6, No. 11
Copyright 2013 VLDB Endowment 2150-8097/13/09... $ 10.00.

- F1 users make heavy use of batching, parallelism and asynchronous reads. We use a new ORM (object-relational mapping) library that makes these concepts explicit. This places an upper bound on the number of RPCs required for typical application-level operations, making those operations scale well by default.

The F1 system has been managing all AdWords advertising campaign data in production since early 2012. AdWords is a vast and diverse ecosystem including 100s of applications and 1000s of users, all sharing the same database. This database is over 100 TB, serves up to hundreds of thousands of requests per second, and runs SQL queries that scan tens of trillions of data rows per day. Availability reaches five nines, even in the presence of unplanned outages, and observable latency on our web applications has not increased compared to the old MySQL system.

We discuss the AdWords F1 database throughout this paper as it was the original and motivating user for F1. Several other groups at Google are now beginning to deploy F1.

2. BASIC ARCHITECTURE

Users interact with F1 through the F1 *client* library. Other tools like the command-line ad-hoc SQL shell are implemented using the same client. The client sends requests to one of many F1 *servers*, which are responsible for reading and writing data from remote data sources and coordinating query execution. Figure 1 depicts the basic architecture and the communication between components.

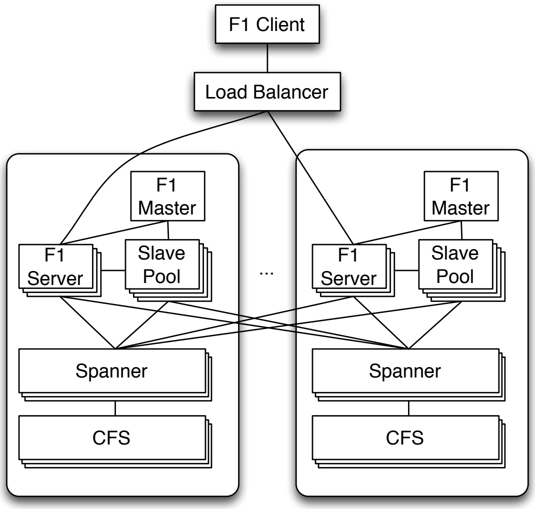

Figure 1: The basic architecture of the F1 system, with servers in two datacenters.

Because of F1's distributed architecture, special care must be taken to avoid unnecessarily increasing request latency. For example, the F1 client and load balancer prefer to connect to an F1 server in a nearby datacenter whenever possible. However, requests may transparently go to F1 servers in remote datacenters in cases of high load or failures.

F1 servers are typically co-located in the same set of datacenters as the Spanner servers storing the data. This co-location ensures that F1 servers generally have fast access to the underlying data. For availability and load balancing, F1 servers can communicate with Spanner servers outside their own datacenter when necessary. The Spanner servers in each datacenter in turn retrieve their data from the Colossus File System (CFS) [14] in the same datacenter. Unlike Spanner, CFS is not a globally replicated service and therefore Spanner servers will never communicate with remote CFS instances.

F1 servers are mostly stateless, allowing a client to communicate with a different F1 server for each request. The one exception is when a client uses pessimistic transactions and must hold locks. The client is then bound to one F1 server for the duration of that transaction. F1 transactions are described in more detail in Section 5. F1 servers can be quickly added (or removed) from our system in response to the total load because F1 servers do not own any data and hence a server addition (or removal) requires no data movement.

An F1 cluster has several additional components that allow for the execution of distributed SQL queries. Distributed execution is chosen over centralized execution when the query planner estimates that increased parallelism will reduce query processing latency. The shared *slave pool* consists of F1 processes that exist only to execute parts of distributed query plans on behalf of regular F1 servers. Slave pool membership is maintained by the F1 *master*, which monitors slave process health and distributes the list of available slaves to F1 servers. F1 also supports large-scale data processing through Google's MapReduce framework [10]. For performance reasons, MapReduce workers are allowed to communicate directly with Spanner servers to extract data in bulk (not shown in the figure). Other clients perform reads and writes exclusively through F1 servers.

The throughput of the entire system can be scaled up by adding more Spanner servers, F1 servers, or F1 slaves. Since F1 servers do not store data, adding new servers does not involve any data re-distribution costs. Adding new Spanner servers results in data re-distribution. This process is completely transparent to F1 servers (and therefore F1 clients).

The Spanner-based remote storage model and our geographically distributed deployment leads to latency characteristics that are very different from those of regular databases. Because the data is synchronously replicated across multiple datacenters, and because we've chosen widely distributed datacenters, the commit latencies are relatively high (50-150 ms). This high latency necessitates changes to the patterns that clients use when interacting with the database. We describe these changes in Section 7.1, and we provide further detail on our deployment choices, and the resulting availability and latency, in Sections 9 and 10.

2.1 Spanner

F1 is built on top of Spanner. Both systems were developed at the same time and in close collaboration. Spanner handles lower-level storage issues like persistence, caching, replication, fault tolerance, data sharding and movement, location lookups, and transactions.

In Spanner, data rows are partitioned into clusters called *directories* using ancestry relationships in the schema. Each directory has at least one *fragment*, and large directories can have multiple fragments. *Groups* store a collection of directory fragments. Each group typically has one replica *tablet* per datacenter. Data is replicated synchronously using the *Paxos* algorithm [18], and all tablets for a group store the same data. One replica tablet is elected as the Paxos *leader* for the group, and that leader is the entry point for all transactional activity for the group. Groups may also include *readonly replicas*, which do not vote in the Paxos algorithm and cannot become the group leader.

Spanner provides serializable pessimistic transactions using strict two-phase locking. A transaction includes multiple reads, taking shared or exclusive locks, followed by a single write that upgrades locks and atomically commits the transaction. All commits are synchronously replicated using Paxos. Transactions are most efficient when updating data co-located in a single group. Spanner also supports transactions across multiple groups, called transaction *participants*, using a two-phase commit (2PC) protocol on top of Paxos. 2PC adds an extra network round trip so it usually doubles observed commit latency. 2PC scales well up to 10s of participants, but abort frequency and latency increase significantly with 100s of participants [7].

Spanner has very strong consistency and timestamp semantics. Every transaction is assigned a commit timestamp, and these timestamps provide a global total ordering for commits. Spanner uses a novel mechanism to pick globally ordered timestamps in a scalable way using hardware clocks deployed in Google datacenters. Spanner uses these timestamps to provide multi-versioned consistent reads, including snapshot reads of current data, without taking read locks. For guaranteed non-blocking, globally consistent reads, Spanner provides a *global safe timestamp*, below which no in-flight or future transaction can possibly commit. The global safe timestamp typically lags current time by 5-10 seconds. Reads at this timestamp can normally run on any replica tablet, including readonly replicas, and they never block behind running transactions.

3. DATA MODEL

3.1 Hierarchical Schema

The F1 data model is very similar to the Spanner data model. In fact, Spanner's original data model was more like Bigtable, but Spanner later adopted F1's data model. At the logical level, F1 has a relational schema similar to that of a traditional RDBMS, with some extensions including explicit table hierarchy and columns with Protocol Buffer data types.

Logically, tables in the F1 schema can be organized into a *hierarchy*. Physically, F1 stores each child table *clustered* with and *interleaved* within the rows from its parent table. Tables from the logical schema cannot be arbitrarily interleaved: the child table must have a foreign key to its parent table as a prefix of its primary key. For example, the AdWords schema contains a table Customer with primary key `(CustomerId)`, which has a child table Campaign with primary key `(CustomerId, CampaignId)`, which in turn has a child table AdGroup with primary key `(CustomerId, CampaignId, AdGroupId)`. A row of the root table in the hierarchy is called a *root row*. All child table rows corresponding to a root row are clustered together with that root row in a single Spanner *directory*, meaning that cluster is normally stored on a single Spanner server. Child rows are stored under their parent row ordered by primary key. Figure 2 shows an example.

The hierarchically clustered physical schema has several advantages over a flat relational schema. Consider the corresponding traditional schema, also depicted in Figure 2. In this traditional schema, fetching all Campaign and AdGroup records corresponding to a given `CustomerId` would take two sequential steps, because there is no direct way to retrieve AdGroup records by `CustomerId`. In the F1 version of the schema, the hierarchical primary keys allow the fetches of Campaign and AdGroup records to be started in parallel, because both tables are keyed by `CustomerId`. The primary key prefix property means that reading all AdGroups for a particular Customer can be expressed as a single range read, rather than reading each row individually using an index. Furthermore, because the tables are both stored in primary key order, rows from the two tables can be joined using a simple ordered merge. Because the data is clustered into a single directory, we can read it all in a single Spanner request. All of these properties of a hierarchical schema help mitigate the latency effects of having remote data.

Hierarchical clustering is especially useful for updates, since it reduces the number of Spanner groups involved in a transaction. Because each root row and all of its descendant rows are stored in a single Spanner directory, transactions restricted to a single root will usually avoid 2PC and the associated latency penalty, so most applications try to use single-root transactions as much as possible. Even when doing transactions across multiple roots, it is important to limit the number of roots involved because adding more participants generally increases latency and decreases the likelihood of a successful commit.

Hierarchical clustering is not mandatory in F1. An F1 schema often has several root tables, and in fact, a completely flat MySQL-style schema is still possible. Using hierarchy however, to the extent that it matches data semantics, is highly beneficial. In AdWords, most transactions are typically updating data for a single advertiser at a time, so we made the advertiser a root table (Customer) and clustered related tables under it. This clustering was critical to achieving acceptable latency.

3.2 Protocol Buffers

The F1 data model supports table columns that contain structured data types. These structured types use the schema and binary encoding format provided by Google's open source Protocol Buffer [16] library. Protocol Buffers have typed fields that can be required, optional, or repeated; fields can also be nested Protocol Buffers. At Google, Protocol Buffers are ubiquitous for data storage and interchange between applications. When we still had a MySQL schema, users often had to write tedious and error-prone transformations between database rows and in-memory data structures. Putting protocol buffers in the schema removes this impedance mismatch and gives users a universal data structure they can use both in the database and in application code.

Protocol Buffers allow the use of *repeated fields*. In F1 schema designs, we often use repeated fields instead of child tables when the number of child records has a low upper bound. By using repeated fields, we avoid the performance overhead and complexity of storing and joining multiple child records. The entire protocol buffer is effectively treated as one blob by Spanner. Aside from performance impacts, Protocol Buffer columns are more natural and reduce semantic complexity for users, who can now read and write their logical business objects as atomic units, without having to

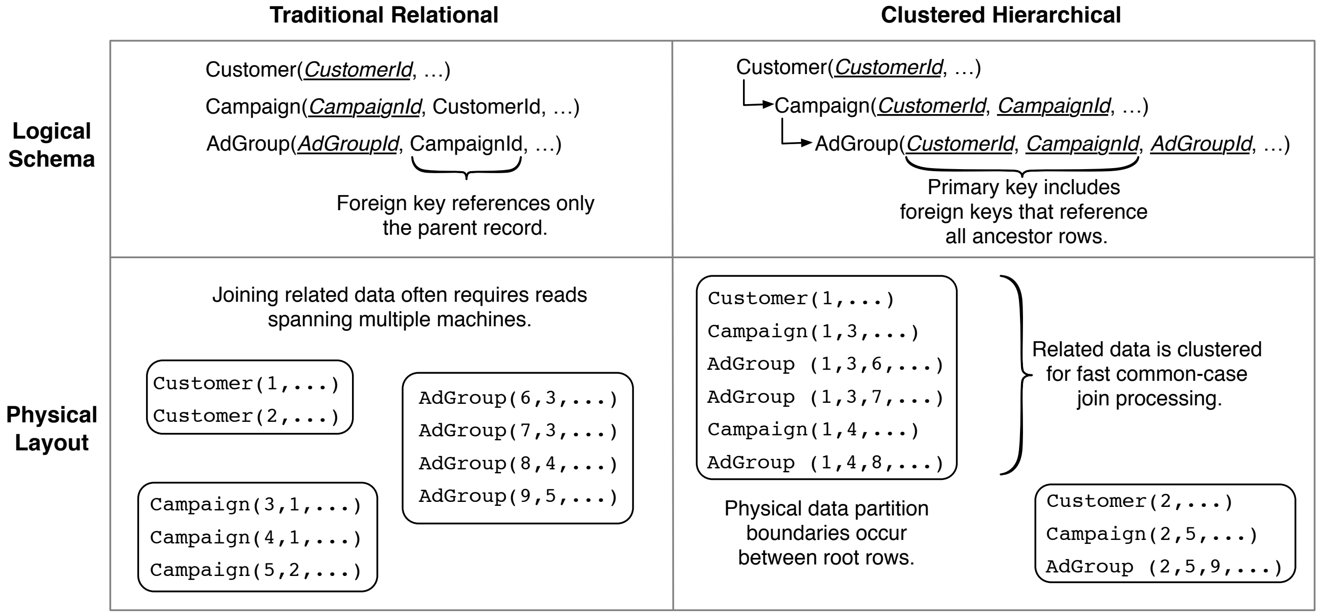
Traditional Relational
Clustered Hierarchical
Logical Schema
Customer(CustomerId, …)
Campaign(CampaignId, CustomerId, …)
AdGroup(AdGroupId, CampaignId, …)
Foreign key references only the parent record.
Customer(CustomerId, …)
Campaign(CustomerId, CampaignId, …)
AdGroup(CustomerId, CampaignId, AdGroupId, …)
Primary key includes foreign keys that reference all ancestor rows.
Physical Layout
Joining related data often requires reads spanning multiple machines.
Customer(1,...)
Customer(2,...)
AdGroup(6,3,...)
AdGroup(7,3,...)
AdGroup(8,4,...)
AdGroup(9,5,...)
Campaign(3,1,...)
Campaign(4,1,...)
Campaign(5,2,...)
Customer(1,...)
Campaign(1,3,...)
AdGroup (1,3,6,...)
AdGroup (1,3,7,...)
Campaign(1,4,...)
AdGroup (1,4,8,...)
Related data is clustered for fast common-case join processing.
Physical data partition boundaries occur between root rows.
Customer(2,...)
Campaign(2,5,...)
AdGroup (2,5,9,...)

Figure 2: The logical and physical properties of data storage in a traditional normalized relational schema compared with a clustered hierarchical schema used in an F1 database.

think about materializing them using joins across several tables. The use of Protocol Buffers in F1 SQL is described in Section 8.7.

Many tables in an F1 schema consist of just a single Protocol Buffer column. Other tables split their data across a handful of columns, partitioning the fields according to access patterns. Tables can be partitioned into columns to group together fields that are usually accessed together, to separate fields with static and frequently updated data, to allow specifying different read/write permissions per column, or to allow concurrent updates to different columns. Using fewer columns generally improves performance in Spanner where there can be high per-column overhead.

3.3 Indexing

All indexes in F1 are transactional and fully consistent. Indexes are stored as separate tables in Spanner, keyed by a concatenation of the index key and the indexed table's primary key. Index keys can be either scalar columns or fields extracted from Protocol Buffers (including repeated fields). There are two types of physical storage layout for F1 indexes: local and global.

Local index keys must contain the root row primary key as a prefix. For example, an index on `(CustomerId, Keyword)` used to store unique keywords for each customer is a local index. Like child tables, local indexes are stored in the same Spanner directory as the root row. Consequently, the index entries of local indexes are stored on the same Spanner server as the rows they index, and local index updates add little additional cost to any transaction.

In contrast, *global index* keys do not include the root row primary key as a prefix and hence cannot be co-located with the rows they index. For example, an index on `(Keyword)` that maps from all keywords in the database to Customers that use them must be global. Global indexes are often large and can have high aggregate update rates. Consequently, they are sharded across many directories and stored on multiple Spanner servers. Writing a single row that updates a global index requires adding a single extra participant to a transaction, which means the transaction must use 2PC, but that is a reasonable cost to pay for consistent global indexes.

Global indexes work reasonably well for single-row updates, but can cause scaling problems for large transactions. Consider a transaction that inserts 1000 rows. Each row requires adding one or more global index entries, and those index entries could be arbitrarily spread across 100s of index directories, meaning the 2PC transaction would have 100s of participants, making it slower and more error-prone. Therefore, we use global indexes sparingly in the schema, and encourage application writers to use small transactions when bulk inserting into tables with global indexes.

Megastore [3] makes global indexes scalable by giving up consistency and supporting only asynchronous global indexes. We are currently exploring other mechanisms to make global indexes more scalable without compromising consistency.

4. SCHEMA CHANGES

The AdWords database is shared by thousands of users and is under constant development. Batches of schema changes are queued by developers and applied daily. This database is mission critical for Google and requires very high availability. Downtime or table locking during schema changes (e.g. adding indexes) is not acceptable.

We have designed F1 to make all schema changes fully non-blocking. Several aspects of the F1 system make non-blocking schema changes particularly challenging:

- F1 is a massively distributed system, with servers in multiple datacenters in distinct geographic regions.

- Each F1 server has a schema locally in memory. It is not practical to make an update occur atomically across all servers.
- Queries and transactions must continue on all tables, even those undergoing schema changes.
- System availability and latency must not be negatively impacted during schema changes.

Because F1 is massively distributed, even if F1 had a global F1 server membership repository, synchronous schema change across all servers would be very disruptive to response times. To make changes atomic, at some point, servers would have to block transactions until confirming all other servers have received the change. To avoid this, F1 schema changes are applied *asynchronously*, on different F1 servers at different times. This implies that two F1 servers may update the database concurrently using different schemas.

If two F1 servers update the database using different schemas that are not compatible according to our schema change algorithms, this could lead to anomalies including database corruption. We illustrate the possibility of database corruption using an example. Consider a schema change from schema S_1 to schema S_2 that adds index I on table T. Because the schema change is applied asynchronously on different F1 servers, assume that server M_1 is using schema S_1 and server M_2 is using schema S_2. First, server M_2 inserts a new row r, which also adds a new index entry $I(r)$ for row r. Subsequently, row r is deleted by server M_1. Because the server is using schema S_1 and is not aware of index I, the server deletes row r, but fails to delete the index entry $I(r)$. Hence, the database becomes corrupt. For example, an index scan on I would return spurious data corresponding to the deleted row r.

We have implemented a schema change algorithm that prevents anomalies similar to the above by

1. Enforcing that across all F1 servers, at most two different schemas are active. Each server uses either the current or next schema. We grant leases on the schema and ensure that no server uses a schema after lease expiry.
2. Subdividing each schema change into multiple phases where consecutive pairs of phases are mutually compatible and cannot cause anomalies. In the above example, we first add index I in a mode where it only executes delete operations. This prohibits server M_1 from adding $I(r)$ into the database. Subsequently, we upgrade index I so servers perform all write operations. Then we initiate a MapReduce to backfill index entries for all rows in table T with carefully constructed transactions to handle concurrent writes. Once complete, we make index I visible for normal read operations.

The full details of the schema change algorithms are covered in [20].

5. TRANSACTIONS

The AdWords product ecosystem requires a data store that supports ACID transactions. We store financial data and have hard requirements on data integrity and consistency. We also have a lot of experience with eventual consistency systems at Google. In all such systems, we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date. We think this is an unacceptable burden to place on developers and that consistency problems should be solved at the database level. Full transactional consistency is one of the most important properties of F1.

Each F1 transaction consists of multiple reads, optionally followed by a single write that commits the transaction. F1 implements three types of transactions, all built on top of Spanner's transaction support:

1. **Snapshot transactions**. These are read-only transactions with snapshot semantics, reading repeatable data as of a fixed Spanner *snapshot timestamp*. By default, snapshot transactions read at Spanner's global safe timestamp, typically 5-10 seconds old, and read from a local Spanner replica. Users can also request a specific timestamp explicitly, or have Spanner pick the current timestamp to see current data. The latter option may have higher latency and require remote RPCs.

   Snapshot transactions are the default mode for SQL queries and for MapReduces. Snapshot transactions allow multiple client servers to see consistent views of the entire database at the same timestamp.

2. **Pessimistic transactions**. These transactions map directly on to Spanner transactions [7]. 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.

3. **Optimistic transactions**. Optimistic transactions consist of a read phase, which can take arbitrarily long and never takes Spanner locks, and then a short write phase. To detect row-level conflicts, F1 returns with each row its last modification timestamp, which is stored in a hidden *lock column* in that row. The new commit timestamp is automatically written into the lock column whenever the corresponding data is updated (in either pessimistic or optimistic transactions). The client library collects these timestamps, and passes them back to an F1 server with the write that commits the transaction. The F1 server creates a short-lived Spanner pessimistic transaction and re-reads the last modification timestamps for all read rows. If any of the re-read timestamps differ from what was passed in by the client, there was a conflicting update, and F1 aborts the transaction. Otherwise, F1 sends the writes on to Spanner to finish the commit.

F1 clients use optimistic transactions by default. Optimistic transactions have several benefits:

- Tolerating misbehaved clients. Reads never hold locks and never conflict with writes. This avoids any problems caused by badly behaved clients who run long transactions or abandon transactions without aborting them.

- Long-lasting transactions. Optimistic transactions can be arbitrarily long, which is useful in some cases. For example, some F1 transactions involve waiting for end-user interaction. It is also hard to debug a transaction that always gets aborted while single-stepping. Idle transactions normally get killed within ten seconds to avoid leaking locks, which means long-running pessimistic transactions often cannot commit.

- Server-side retriability. Optimistic transaction commits are self-contained, which makes them easy to retry transparently in the F1 server, hiding most transient Spanner errors from the user. Pessimistic transactions cannot be retried by F1 servers because they require re-running the user's business logic to reproduce the same locking side-effects.

- Server failover. All state associated with an optimistic transaction is kept on the client. Consequently, the client can send reads and commits to different F1 servers after failures or to balance load.

- Speculative writes. A client may read values outside an optimistic transaction (possibly in a MapReduce), and remember the timestamp used for that read. Then the client can use those values and timestamps in an optimistic transaction to do speculative writes that only succeed if no other writes happened after the original read.

Optimistic transactions do have some drawbacks:

- Insertion phantoms. Modification timestamps only exist for rows present in the table, so optimistic transactions do not prevent insertion phantoms [13]. Where this is a problem, it is possible to use parent-table locks to avoid phantoms. (See Section 5.1)

- Low throughput under high contention. For example, in a table that maintains a counter which many clients increment concurrently, optimistic transactions lead to many failed commits because the read timestamps are usually stale by write time. In such cases, pessimistic transactions with exclusive locks avoid the failed transactions, but also limit throughput. If each commit takes 50ms, at most 20 transactions per second are possible. Improving throughput beyond that point requires application-level changes, like batching updates.

F1 users can mix optimistic and pessimistic transactions arbitrarily and still preserve ACID semantics. All F1 writes update the last modification timestamp on every relevant lock column. Snapshot transactions are independent of any write transactions, and are also always consistent.

5.1 Flexible Locking Granularity

F1 provides row-level locking by default. Each F1 row contains one *default lock column* that covers all columns in the same row. However, concurrency levels can be changed in the schema. For example, users can increase concurrency by defining additional lock columns in the same row, with each lock column covering a subset of columns. In an extreme case, each column can be covered by a separate lock column, resulting in column-level locking.

One common use for column-level locking is in tables with concurrent writers, where each updates a different set of columns. For example, we could have a front-end system allowing users to change bids for keywords, and a back-end system that updates serving history on the same keywords. Busy customers may have continuous streams of bid updates at the same time that back-end systems are updating stats. Column-level locking avoids transaction conflicts between these independent streams of updates.

Users can also selectively reduce concurrency by using a lock column in a parent table to cover columns in a child table. This means that a set of rows in the child table share the same lock column and writes within this set of rows get serialized. Frequently, F1 users use lock columns in parent tables to avoid insertion phantoms for specific predicates or make other business logic constraints easier to enforce. For example, there could be a limit on keyword count per AdGroup, and a rule that keywords must be distinct. Such constraints are easy to enforce correctly if concurrent keyword insertions (in the same AdGroup) are impossible.

6. CHANGE HISTORY

Many database users build mechanisms to log changes, either from application code or using database features like triggers. In the MySQL system that AdWords used before F1, our Java application libraries added change history records into all transactions. This was nice, but it was inefficient and never 100% reliable. Some classes of changes would not get history records, including changes written from Python scripts and manual SQL data changes.

In F1, Change History is a first-class feature at the database level, where we can implement it most efficiently and can guarantee full coverage. In a change-tracked database, all tables are change-tracked by default, although specific tables or columns can be opted out in the schema. Every transaction in F1 creates one or more *ChangeBatch* Protocol Buffers, which include the primary key and before and after values of changed columns for each updated row. These ChangeBatches are written into normal F1 tables that exist as children of each root table. The primary key of the ChangeBatch table includes the associated root table key and the transaction commit timestamp. When a transaction updates data under multiple root rows, possibly from different root table hierarchies, one ChangeBatch is written for each distinct root row (and these ChangeBatches include pointers to each other so the full transaction can be reassembled if necessary). This means that for each root row, the change history table includes ChangeBatches showing all changes associated with children of that root row, in commit order, and this data is easily queryable with SQL. This clustering also means that change history is stored close to the data being tracked, so these additional writes normally do not add additional participants into Spanner transactions, and therefore have minimal latency impact.

F1's ChangeHistory mechanism has a variety of uses. The most common use is in applications that want to be notified of changes and then do some incremental processing. For example, the approval system needs to be notified when new ads have been inserted so it can approve them. F1 uses a publish-and-subscribe system to push notifications that particular root rows have changed. The publish happens in Spanner and is guaranteed to happen at least once after any series of changes to any root row. Subscribers normally remember a checkpoint (i.e. a high-water mark) for each root row and read all changes newer than the checkpoint whenever they receive a notification. This is a good example of a place where Spanner's timestamp ordering properties are very powerful since they allow using checkpoints to guarantee that every change is processed exactly once. A separate system exists that makes it easy for these clients to see only changes to tables or columns they care about.

Change History also gets used in interesting ways for caching. One client uses an in-memory cache based on database state, distributed across multiple servers, and uses this while rendering pages in the AdWords web UI. After a user commits an update, it is important that the next page rendered reflects that update. When this client reads from the cache, it passes in the root row key and the commit timestamp of the last write that must be visible. If the cache is behind that timestamp, it reads Change History records beyond its checkpoint and applies those changes to its in-memory state to catch up. This is much cheaper than reloading the cache with a full extraction and much simpler and more accurate than comparable cache invalidation protocols.

7. CLIENT DESIGN

7.1 Simplified ORM

The nature of working with a distributed data store required us to rethink the way our client applications interacted with the database. Many of our client applications had been written using a MySQL-based ORM layer that could not be adapted to work well with F1. Code written using this library exhibited several common ORM anti-patterns:

- Obscuring database operations from developers.
- Serial reads, including `for` loops that do one query per iteration.
- Implicit traversals: adding unwanted joins and loading unnecessary data "just in case".

Patterns like these are common in ORM libraries. They may save development time in small-scale systems with local data stores, but they hurt scalability even then. When combined with a high-latency remote database like F1, they are disastrous. For F1, we replaced this ORM layer with a new, stripped-down API that forcibly avoids these anti-patterns. The new ORM layer does not use any joins and does not implicitly traverse any relationships between records. All object loading is explicit, and the ORM layer exposes APIs that promote the use of parallel and asynchronous read access. This is practical in an F1 schema for two reasons. First, there are simply fewer tables, and clients are usually loading Protocol Buffers directly from the database. Second, hierarchically structured primary keys make loading all children of an object expressible as a single range read without a join.

With this new F1 ORM layer, application code is more explicit and can be slightly more complex than code using the MySQL ORM, but this complexity is partially offset by the reduced impedance mismatch provided by Protocol Buffer columns. The transition usually results in better client code that uses more efficient access patterns to the database. Avoiding serial reads and other anti-patterns results in code that scales better with larger data sets and exhibits a flatter overall latency distribution.

With MySQL, latency in our main interactive application was highly variable. Average latency was typically 200-300 ms. Small operations on small customers would run much faster than that, but large operations on large customers could be much slower, with a latency tail of requests taking multiple seconds. Developers regularly fought to identify and fix cases in their code causing excessively serial reads and high latency. With our new coding style on the F1 ORM, this doesn't happen. User requests typically require a fixed number (fewer than 10) of reads, independent of request size or data size. The minimum latency is higher than in MySQL because of higher minimum read cost, but the average is about the same, and the latency tail for huge requests is only a few times slower than the median.

7.2 NoSQL Interface

F1 supports a NoSQL key/value based interface that allows for fast and simple programmatic access to rows. Read requests can include any set of tables, requesting specific columns and key ranges for each. Write requests specify inserts, updates, and deletes by primary key, with any new column values, for any set of tables.

This interface is used by the ORM layer under the hood, and is also available for clients to use directly. This API allows for batched retrieval of rows from multiple tables in a single call, minimizing the number of round trips required to complete a database transaction. Many applications prefer to use this NoSQL interface because it's simpler to construct structured read and write requests in code than it is to generate SQL. This interface can be also be used in MapReduces to specify which data to read.

7.3 SQL Interface

F1 also provides a full-fledged SQL interface, which is used for low-latency OLTP queries, large OLAP queries, and everything in between. F1 supports joining data from its Spanner data store with other data sources including Bigtable, CSV files, and the aggregated analytical data warehouse for AdWords. The SQL dialect extends standard SQL with constructs that allow accessing data stored in Protocol Buffers. Updates are also supported using SQL data manipulation statements, with extensions to support updating fields inside protocol buffers and to deal with repeated structure inside protocol buffers. Full syntax details are beyond the scope of this paper.

8. QUERY PROCESSING

The F1 SQL query processing system has the following key properties which we will elaborate on in this section:

- Queries are executed either as low-latency centrally executed queries or distributed queries with high parallelism.
- All data is remote and batching is used heavily to mitigate network latency.
- All input data and internal data is arbitrarily partitioned and has few useful ordering properties.
- Queries use many hash-based repartitioning steps.
- Individual query plan operators are designed to stream data to later operators as soon as possible, maximizing pipelining in query plans.

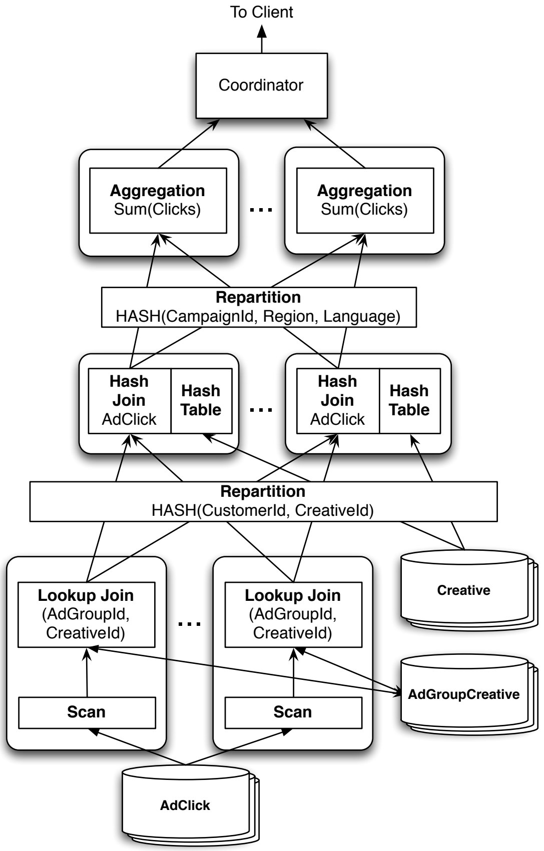

Figure 3: A distributed query plan. Rounded boxes represent processes running on separate machines. Arrows show data flow within a process or over the network in the form of RPCs.

- Hierarchically clustered tables have optimized access methods.
- Query data can be consumed in parallel.
- Protocol Buffer-valued columns provide first-class support for structured data types.
- Spanner's snapshot consistency model provides globally consistent results.

8.1 Central and Distributed Queries

F1 SQL supports both centralized and distributed execution of queries. Centralized execution is used for short OLTP-style queries and the entire query runs on one F1 server node. Distributed execution is used for OLAP-style queries and spreads the query workload over worker tasks in the F1 slave pool (see Section 2). Distributed queries always use snapshot transactions. The query optimizer uses heuristics to determine which execution mode is appropriate for a given query. In the sections that follow, we will mainly focus our attention on distributed query execution. Many of the concepts apply equally to centrally executed queries.

8.2 Distributed Query Example

The following example query would be answered using distributed execution:

```
SELECT agcr.CampaignId, click.Region,
       cr.Language, SUM(click.Clicks)
FROM AdClick click
  JOIN AdGroupCreative agcr
    USING (AdGroupId, CreativeId)
  JOIN Creative cr
    USING (CustomerId, CreativeId)
WHERE click.Date = '2013-03-23'
GROUP BY agcr.CampaignId, click.Region,
         cr.Language
```

This query uses part of the AdWords schema. An *AdGroup* is a collection of ads with some shared configuration. A *Creative* is the actual ad text. The *AdGroupCreative* table is a link table between AdGroup and Creative; Creatives can be shared by multiple AdGroups. Each *AdClick* records the Creative that the user was shown and the AdGroup from which the Creative was chosen. This query takes all AdClicks on a specific date, finds the corresponding AdGroupCreative and then the Creative. It then aggregates to find the number of clicks grouped by campaign, region and language.

A possible query plan for this query is shown in Figure 3. In the query plan, data is streamed bottom-up through each of the operators up until the aggregation operator. The deepest operator performs a scan of the AdClick table. In the same worker node, the data from the AdClick scan flows into a *lookup join* operator, which looks up AdGroupCreative records using a secondary index key. The plan then *repartitions* the data stream by a hash of the CustomerId and CreativeId, and performs a lookup in a hash table that is partitioned in the same way (a distributed hash join). After the distributed hash join, the data is once again repartitioned, this time by a hash of the CampaignId, Region and Language fields, and then fed into an aggregation operator that groups by those same fields (a distributed aggregation).

8.3 Remote Data

SQL query processing, and join processing in particular, poses some interesting challenges in F1, primarily because F1 does not store its data locally. F1's main data store is Spanner, which is a remote data source, and F1 SQL can also access other remote data sources and join across them. These remote data accesses involve highly variable network latency [9]. In contrast, traditional database systems generally perform processing on the same machine that hosts their data, and they mostly optimize to reduce the number of disk seeks and disk accesses.

Network latency and disk latency are fundamentally different in two ways. First, network latency can be mitigated by *batching* or *pipelining* data accesses. F1 uses extensive batching to mitigate network latency. Secondly, disk latency is generally caused by contention for a single limited resource, the actual disk hardware. This severely limits the usefulness of sending out multiple data accesses at the same time. In contrast, F1's network based storage is typically distributed over many disks, because Spanner partitions its data across many physical servers, and also at a finer-grained level because Spanner stores its data in CFS. This makes it much less likely that multiple data accesses will contend for the same resources, so scheduling multiple data accesses in parallel often results in near-linear speedup until the underlying storage system is truly overloaded.

The prime example of how F1 SQL takes advantage of batching is found in the *lookup join* query plan operator. This operator executes a join by reading from the inner table using equality lookup keys. It first retrieves rows from the outer table, extracting the lookup key values from them and deduplicating those keys. This continues until it has gathered 50MB worth of data or 100,000 unique lookup key values. Then it performs a simultaneous lookup of all keys in the inner table. This returns the requested data in arbitrary order. The lookup join operator joins the retrieved inner table rows to the outer table rows which are stored in memory, using a hash table for fast lookup. The results are streamed immediately as output from the lookup join node.

F1's query operators are designed to stream data as much as possible, reducing the incidence of pipeline stalls. This design decision limits operators' ability to preserve interesting data orders. Specifically, an F1 operator often has many reads running asynchronously in parallel, and streams rows to the next operator as soon as they are available. This emphasis on data streaming means that ordering properties of the input data are lost while allowing for maximum read request concurrency and limiting the space needed for row buffering.

8.4 Distributed Execution Overview

The structure of a distributed query plan is as follows. A full query plan consists of potentially tens of *plan parts*, each of which represents a number of workers that execute the same query subplan. The plan parts are organized as a directed acyclic graph (DAG), with data flowing up from the leaves of the DAG to a single *root node*, which is the only node with no out edges, i.e. the only sink. The root node, also called the *query coordinator*, is executed by the server that received the incoming SQL query request from a client. The query coordinator plans the query for execution, receives results from the penultimate plan parts, performs any final aggregation, sorting, or filtering, and then streams the results back to the client, except in the case of partitioned consumers as described in Section 8.6.

A technique frequently used by distributed database systems is to take advantage of an explicit co-partitioning of the stored data. Such co-partitioning can be used to push down large amounts of query processing onto each of the processing nodes that host the partitions. F1 cannot take advantage of such partitioning, in part because the data is always remote, but more importantly, because Spanner applies an arbitrary, effectively random partitioning. Moreover, Spanner can also dynamically change the partitioning. Hence, to perform operations efficiently, F1 must frequently resort to repartitioning the data. Because none of the input data is range partitioned, and because range partitioning depends on correct statistics, we have eschewed range partitioning altogether and opted to only apply hash partitioning.

Traditionally, repartitioning like this has been regarded as something to be avoided because of the heavy network traffic involved. Recent advances in the scalability of network switch hardware have allowed us to connect clusters of several hundred F1 worker processes in such a way that all servers can simultaneously communicate with each other at close to full network interface speed. This allows us to repartition without worrying much about network capacity and concepts like rack affinity. A potential downside of this solution is that it limits the size of an F1 cluster to the limits of available network switch hardware. This has not posed a problem in practice for the queries and data sizes that the F1 system deals with.

The use of hash partitioning allows us to implement an efficient distributed hash join operator and a distributed aggregation operator. These operators were already demonstrated in the example query in Section 8.2. The hash join operator repartitions both of its inputs by applying a hash function to the join keys. In the example query, the hash join keys are CustomerId and CreativeId. Each worker is responsible for a single partition of the hash join. Each worker loads its smallest input (as estimated by the query planner) into an in-memory hash table. It then reads its largest input and probes the hash table for each row, streaming out the results. For distributed aggregation, we aggregate as much as possible locally inside small buffers, then repartition the data by a hash of the grouping keys, and finally perform a full aggregation on each of the hash partitions. When hash tables grow too large to fit in memory, we apply standard algorithms that spill parts of the hash table to disk.

F1 SQL operators execute in memory, without checkpointing to disk, and stream data as much as possible. This avoids the cost of saving intermediate results to disk, so queries run as fast as the data can be processed. This means, however, that any server failure can make an entire query fail. Queries that fail get retried transparently, which usually hides these failures. In practice, queries that run for up to an hour are sufficiently reliable, but queries much longer than that may experience too many failures. We are exploring adding checkpointing for some intermediate results into our query plans, but this is challenging to do without hurting latency in the normal case where no failures occur.

8.5 Hierarchical Table Joins

As described in Section 3.1, the F1 data model supports hierarchically clustered tables, where the rows of a child table are interleaved in the parent table. This data model allows us to efficiently join a parent table and a descendant table by their shared primary key prefix. For instance, consider the join of table `Customer` with table `Campaign`:

```
SELECT *
FROM Customer JOIN
     Campaign USING (CustomerId)
```

The hierarchically clustered data model allows F1 to perform this join using a *single* request to Spanner in which we request the data from both tables. Spanner will return the data to F1 in interleaved order (a pre-order depth-first traversal), ordered by primary key prefix, e.g.:

```
Customer(3)
  Campaign(3,5)
  Campaign(3,6)
Customer(4)
  Campaign(4,2)
  Campaign(4,4)
```

While reading this stream, F1 uses a merge-join-like algorithm which we call *cluster join*. The cluster join operator only needs to buffer 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 cluster joined this way using a single Spanner request, as long as all tables fall on a single ancestry path in the table hierarchy.

For instance, in the following table hierarchy, F1 SQL can only join RootTable to either ChildTable1 or ChildTable2 in this way, but not both:

```
RootTable
  ChildTable1
  ChildTable2
```

When F1 SQL has to join between sibling tables like these, it will perform one join operation using the cluster join algorithm, and select an alternate join algorithm for the remaining join. An algorithm such as lookup join is able to perform this join without disk spilling or unbounded memory usage because it can construct the join result piecemeal using bounded-size batches of lookup keys.

8.6 Partitioned Consumers

F1 queries can produce vast amounts of data, and pushing this data through a single query coordinator can be a bottleneck. Furthermore, a single client process receiving all the data can also be a bottleneck and likely cannot keep up with many F1 servers producing result rows in parallel. To solve this, F1 allows multiple client processes to consume sharded streams of data from the same query in parallel. This feature is used for partitioned consumers like MapReduces[10]. The client application sends the query to F1 and requests distributed data retrieval. F1 then returns a set of endpoints to connect to. The client must connect to all of these endpoints and retrieve the data in parallel. Due to the streaming nature of F1 queries, and the cross-dependencies caused by frequent hash repartitioning, slowness in one distributed reader may slow other distributed readers as well, as the F1 query produces results for all readers in lock-step. A possible, but so far unimplemented mitigation strategy for this horizontal dependency is to use disk-backed buffering to break the dependency and to allow clients to proceed independently.

8.7 Queries with Protocol Buffers

As explained in Section 3, the F1 data model makes heavy use of Protocol Buffer valued columns. The F1 SQL dialect treats these values as first class objects, providing full access to all of the data contained therein. For example, the following query requests the CustomerId and the entire Protocol Buffer valued column Info for each customer whose country code is US.

```
SELECT c.CustomerId, c.Info
FROM Customer AS c
WHERE c.Info.country_code = 'US'
```

This query illustrates two aspects of Protocol Buffer support. First, queries use path expressions to extract individual fields (c.Info.country_code). Second, F1 SQL also allows for querying and passing around entire protocol buffers (c.Info). Support for full Protocol Buffers reduces the impedance mismatch between F1 SQL and client applications, which often prefer to receive complete Protocol Buffers.

Protocol Buffers also allow *repeated fields*, which may have zero or more instances, i.e., they can be regarded as variable-length arrays. When these repeated fields occur in F1 database columns, they are actually very similar to hierarchical child tables in a 1:N relationship. The main difference between a child table and a repeated field is that the child table contains an explicit foreign key to its parent table, while the repeated field has an *implicit* foreign key to the Protocol Buffer containing it. Capitalizing on this similarity, F1 SQL supports access to repeated fields using *PROTO JOIN*, a JOIN variant that joins by the implicit foreign key. For instance, suppose that we have a table Customer, which has a Protocol Buffer column Whitelist which in turn contains a repeated field feature. Furthermore, suppose that the values of this field feature are themselves Protocol Buffers, each of which represents the whitelisting status of a particular feature for the parent Customer.

```
SELECT c.CustomerId, f.feature
FROM Customer AS c
     PROTO JOIN c.Whitelist.feature AS f
WHERE f.status = 'STATUS_ENABLED'
```

This query joins the Customer table with its virtual child table Whitelist.feature by the foreign key that is implied by containment. It then filters the resulting combinations by the value of a field f.status inside the child table f, and returns another field f.feature from that child table. In this query syntax, the PROTO JOIN specifies the parent relation of the repeated field by qualifying the repeated field name with c, which is the alias of the parent relation. The implementation of the PROTO JOIN construct is straightforward: in the read from the outer relation we retrieve the entire Protocol Buffer column containing the repeated field, and then for each outer row we simply enumerate the repeated field instances in memory and join them to the outer row.

F1 SQL also allows subqueries on repeated fields in Protocol Buffers. The following query has a scalar subquery to count the number of Whitelist.features, and an EXISTS subquery to select only Customers that have at least one feature that is not ENABLED. Each subquery iterates over repeated field values contained inside Protocol Buffers from the current row.

```
SELECT c.CustomerId, c.Info,
  (SELECT COUNT(*) FROM c.Whitelist.feature) nf
FROM Customer AS c
WHERE EXISTS (SELECT * FROM c.Whitelist.feature f
              WHERE f.status != 'ENABLED')
```

Protocol Buffers have performance implications for query processing. First, we always have to fetch entire Protocol Buffer columns from Spanner, even when we are only interested in a small subset of fields. This takes both additional network and disk bandwidth. Second, in order to extract the fields that the query refers to, we always have to parse the contents of the Protocol Buffer fields. Even though we have implemented an optimized parser to extract only requested fields, the impact of this decoding step is significant. Future versions will improve this by pushing parsing and field selection to Spanner, thus reducing network bandwidth required and saving CPU in F1 while possibly using more CPU in Spanner.

9. DEPLOYMENT

The F1 and Spanner clusters currently deployed for AdWords use five datacenters spread out across mainland US. The Spanner configuration uses 5-way Paxos replication to ensure high availability. Each region has additional read-only replicas that do not participate in the Paxos algorithm. Read-only replicas are used only for snapshot reads and thus allow us to segregate OLTP and OLAP workloads.

Intuitively, 3-way replication should suffice for high availability. In practice, this is not enough. When one datacenter is down (because of either an outage or planned maintenance), both surviving replicas must remain available for F1 to be able to commit transactions, because a Paxos commit must succeed on a majority of replicas. If a second datacenter goes down, the entire database becomes completely unavailable. Even a single machine failure or restart temporarily removes a second replica, causing unavailability for data hosted on that server.

Spanner's Paxos implementation designates one of the replicas as a *leader*. All transactional reads and commits must be routed to the leader replica. User transactions normally require at least two round trips to the leader (reads followed by a commit). Moreover, F1 servers usually perform an extra read as a part of transaction commit (to get old values for Change History, index updates, optimistic transaction timestamp verification, and referential integrity checks). Consequently, transaction latency is best when clients and F1 servers are co-located with Spanner leader replicas. We designate one of the datacenters as a *preferred leader location*. Spanner locates leader replicas in the preferred leader location whenever possible. Clients that perform heavy database modifications are usually deployed close to the preferred leader location. Other clients, including those that primarily run queries, can be deployed anywhere, and normally do reads against local F1 servers.

We have chosen to deploy our five read/write replicas with two each on the east and west coasts of the US, and the fifth centrally. With leaders on the east coast, commits require round trips to the other east coast datacenter, plus the central datacenter, which accounts for the 50ms minimum latency. We have chosen this deployment to maximize availability in the presence of large regional outages. Other F1 and Spanner instances could be deployed with closer replicas to reduce commit latency.

10. LATENCY AND THROUGHPUT

In our configuration, F1 users see read latencies of 5-10 ms and commit latencies of 50-150 ms. Commit latency is largely determined by network latency between datacenters. The Paxos algorithm allows a transaction to commit once a majority of voting Paxos replicas acknowledge the transaction. With five replicas, commits require a round trip from the leader to the two nearest replicas. Multi-group commits require 2PC, which typically doubles the minimum latency.

Despite the higher database latency, overall user-facing latency for the main interactive AdWords web application averages about 200ms, which is similar to the preceding system running on MySQL. Our schema clustering and application coding strategies have successfully hidden the inherent latency of synchronous commits. Avoiding serial reads in client code accounts for much of that. In fact, while the average is similar, the MySQL application exhibited tail latency much worse than the same application on F1.

For non-interactive applications that apply bulk updates, we optimize for throughput rather than latency. We typically structure such applications so they do small transactions, scoped to single Spanner directories when possible, and use parallelism to achieve high throughput. For example, we have one application that updates billions of rows per day, and we designed it to perform single-directory transactions of up to 500 rows each, running in parallel and aiming for 500 transactions per second. F1 and Spanner provide very high throughput for parallel writes like this and are usually not a bottleneck – our rate limits are usually chosen to protect downstream Change History consumers who can't process changes fast enough.

For query processing, we have mostly focused on functionality and parity so far, and not on absolute query performance. Small central queries reliably run in less than 10ms, and some applications do tens or hundreds of thousands of SQL queries per second. Large distributed queries run with latency comparable to MySQL. Most of the largest queries actually run faster in F1 because they can use more parallelism than MySQL, which can only parallelize up to the number of MySQL shards. In F1, such queries often see linear speedup when given more resources.

Resource costs are usually higher in F1, where queries often use an order of magnitude more CPU than similar MySQL queries. MySQL stored data uncompressed on local disk, and was usually bottlenecked by disk rather than CPU, even with flash disks. F1 queries start from data compressed on disk and go through several layers, decompressing, processing, recompressing, and sending over the network, all of which have significant cost. Improving CPU efficiency here is an area for future work.

11. RELATED WORK

As a hybrid of relational and NoSQL systems, F1 is related to work in both areas. F1's relational query execution techniques are similar to those described in the shared-nothing database literature, e.g., [12], with some key differences like the ignoring of interesting orders and the absence of co-partitioned data. F1's NoSQL capabilities share properties with other well-described scalable key-value stores including Bigtable [6], HBase [1], and Dynamo [11]. The hierarchical schema and clustering properties are similar to Megastore [3].

Optimistic transactions have been used in previous systems including Percolator [19] and Megastore [3]. There is extensive literature on transaction, consistency and locking models, including optimistic and pessimistic transactions, such as [24] and [4].

Prior work [15] also chose to mitigate the inherent latency of remote lookups though the use of asynchrony in query processing. However, due to the large volumes of data processed by F1, our system is not able to make the simplifying assumption that an unlimited number of asynchronous requests can be made at the same time. This complication, coupled with the high variability of storage operation latency, led to the out-of-order streaming design described in Section 8.3.

MDCC [17] suggests some Paxos optimizations that could be applied to reduce the overhead of multi-participant transactions.

Using Protocol Buffers as first class types makes F1, in part, a kind of object database [2]. The resulting simplified ORM results in a lower impedance mismatch for most client applications at Google, where Protocol Buffers are used pervasively. The hierarchical schema and clustering properties are similar to Megastore and ElasTraS [8]. F1 treats repeated fields inside protocol buffers like nested relations [21].

12. CONCLUSION

In recent years, conventional wisdom in the engineering community has been that if you need a highly scalable, high-throughput data store, the only viable option is to use a NoSQL key/value store, and to work around the lack of ACID transactional guarantees and the lack of conveniences like secondary indexes, SQL, and so on. When we sought a replacement for Google's MySQL data store for the AdWords product, that option was simply not feasible: the complexity of dealing with a non-ACID data store in every part of our business logic would be too great, and there was simply no way our business could function without SQL queries. Instead of going NoSQL, we built F1, a distributed relational database system that combines high availability, the throughput and scalability of NoSQL systems, and the functionality, usability and consistency of traditional relational databases, including ACID transactions and SQL queries.

Google's core AdWords business is now running completely on F1. F1 provides the SQL database functionality that our developers are used to and our business requires. Unlike our MySQL solution, F1 is trivial to scale up by simply adding machines. Our low-level commit latency is higher, but by using a coarse schema design with rich column types and improving our client application coding style, the observable end-user latency is as good as before and the worst-case latencies have actually improved.

F1 shows that it is actually possible to have a highly scalable and highly available distributed database that still provides all of the guarantees and conveniences of a traditional relational database.

13. ACKNOWLEDGEMENTS

We would like to thank the Spanner team, without whose great efforts we could not have built F1. We'd also like to thank the many developers and users across all AdWords teams who migrated their systems to F1, and who played a large role influencing and validating the design of this system. We also thank all former and newer F1 team members, including Michael Armbrust who helped write this paper, and Marcel Kornacker who worked on the early design of the F1 query engine.

14. REFERENCES

[1] Apache Foundation. Apache HBase. `http://hbase.apache.org/`.

[2] M. Atkinson et al. The object-oriented database system manifesto. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an object-oriented database system*, pages 1–20. Morgan Kaufmann, 1992.

[3] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[4] H. Berenson et al. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[5] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC*, 2000.

[6] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[8] S. Das et al. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *TODS*, 38(1):5:1–5:45, Apr. 2013.

[9] J. Dean. Evolution and future directions of large-scale storage and computation systems at Google. In *SOCC*, 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[12] D. J. Dewitt et al. The Gamma database machine project. *TKDE*, 2(1):44–62, Mar. 1990.

[13] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, Nov. 1976.

[14] A. Fikes. Storage architecture and challenges. Google Faculty Summit, July 2010.

[15] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.

[16] Google, Inc. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[17] T. Kraska et al. MDCC: Multi-data center consistency. In *EuroSys*, 2013.

[18] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[19] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[20] I. Rae et al. Online, asynchronous schema change in F1. *PVLDB*, 6(11), 2013.

[21] M. A. Roth et al. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13(4):389–417, Oct. 1988.

[22] J. Shute et al. F1: The fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD*, 2012.

[23] M. Stonebraker. SQL databases v. NoSQL databases. *CACM*, 53(4), 2010.

[24] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.