Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

provides externally consistent [16] reads and writes, and globally-consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent MapReduce executions [12], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

These features are enabled by the fact that Spanner assigns globally-meaningful commit timestamps to transactions, even though transactions may be distributed. The timestamps reflect serialization order. In addition, the serialization order satisfies external consistency (or equivalently, linearizability [20]): if a transaction T_1 commits before another transaction T_2 starts, then T_1's commit timestamp is smaller than T_2's. Spanner is the first system to provide such guarantees at global scale.

The key enabler of these properties is a new TrueTime API and its implementation. The API directly exposes clock uncertainty, and the guarantees on Spanner's timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty. Google's cluster-management software provides an implementation of the TrueTime API. This implementation keeps uncertainty small (generally less than 10ms) by using multiple modern clock references (GPS and atomic clocks).

Section 2 describes the structure of Spanner's implementation, its feature set, and the engineering decisions that went into their design. Section 3 describes our new TrueTime API and sketches its implementation. Section 4 describes how Spanner uses TrueTime to implement externally-consistent distributed transactions, lock-free read-only transactions, and atomic schema updates. Section 5 provides some benchmarks on Spanner's performance and TrueTime behavior, and discusses the experiences of F1. Sections 6, 7, and 8 describe related and future work, and summarize our conclusions.

2 Implementation

This section describes the structure of and rationale underlying Spanner's implementation. It then describes the *directory* abstraction, which is used to manage replication and locality, and is the unit of data movement. Finally, it describes our data model, why Spanner looks like a relational database instead of a key-value store, and how applications can control data locality.

A Spanner deployment is called a *universe*. Given that Spanner manages data globally, there will be only a handful of running universes. We currently run a test/playground universe, a development/production universe, and a production-only universe.

Spanner is organized as a set of *zones*, where each zone is the rough analog of a deployment of Bigtable servers [9]. Zones are the unit of administrative deployment. The set of zones is also the set of locations across which data can be replicated. Zones can be added to or removed from a running system as new datacenters are brought into service and old ones are turned off, respectively. Zones are also the unit of physical isolation: there may be one or more zones in a datacenter, for example, if different applications' data must be partitioned across different sets of servers in the same datacenter.

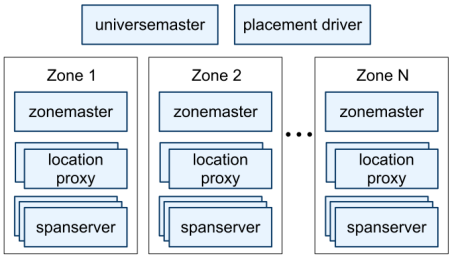

Figure 1: Spanner server organization.

Figure 1 illustrates the servers in a Spanner universe. A zone has one *zonemaster* and between one hundred and several thousand *spanservers*. The former assigns data to spanservers; the latter serve data to clients. The per-zone *location proxies* are used by clients to locate the spanservers assigned to serve their data. The *universe master* and the *placement driver* are currently singletons. The universe master is primarily a console that displays status information about all the zones for interactive debugging. The placement driver handles automated movement of data across zones on the timescale of minutes. The placement driver periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load. For space reasons, we will only describe the spanserver in any detail.

2.1 Spanserver Software Stack

This section focuses on the spanserver implementation to illustrate how replication and distributed transactions have been layered onto our Bigtable-based implementation. The software stack is shown in Figure 2. At the bottom, each spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*. A tablet is similar to Bigtable's tablet abstraction, in that it implements a bag of the following mappings:

$$(\text{key:string}, \text{timestamp:int64}) \rightarrow \text{string}$$

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multi-version database than a key-value store. A

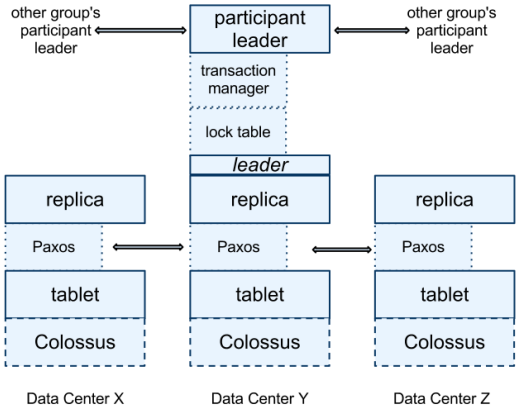

Figure 2: Spanserver software stack.

tablet's state is stored in set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System [15]).

To support replication, each spanserver implements a single Paxos state machine on top of each tablet. (An early Spanner incarnation supported multiple Paxos state machines per tablet, which allowed for more flexible replication configurations. The complexity of that design led us to abandon it.) Each state machine stores its metadata and log in its corresponding tablet. Our Paxos implementation supports long-lived leaders with time-based leader leases, whose length defaults to 10 seconds. The current Spanner implementation logs every Paxos write twice: once in the tablet's log, and once in the Paxos log. This choice was made out of expediency, and we are likely to remedy this eventually. Our implementation of Paxos is pipelined, so as to improve Spanner's throughput in the presence of WAN latencies; but writes are applied by Paxos in order (a fact on which we will depend in Section 4).

The Paxos state machines are used to implement a consistently replicated bag of mappings. The key-value mapping state of each replica is stored in its corresponding tablet. Writes must initiate the Paxos protocol at the leader; reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date. The set of replicas is collectively a *Paxos group*.

At every replica that is a leader, each spanserver implements a *lock table* to implement concurrency control. The lock table contains the state for two-phase locking: it maps ranges of keys to lock states. (Note that having a long-lived Paxos leader is critical to efficiently managing the lock table.) In both Bigtable and Spanner, we designed for long-lived transactions (for example, for report generation, which might take on the order of minutes), which perform poorly under optimistic concurrency control in the presence of conflicts. Operations

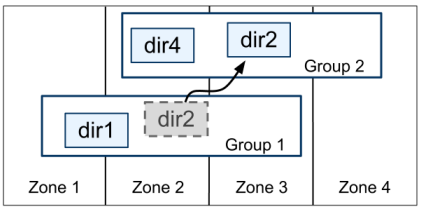

Figure 3: Directories are the unit of data movement between Paxos groups.

that require synchronization, such as transactional reads, acquire locks in the lock table; other operations bypass the lock table.

At every replica that is a leader, each spanserver also implements a *transaction manager* to support distributed transactions. The transaction manager is used to implement a *participant leader*; the other replicas in the group will be referred to as *participant slaves*. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the *coordinator leader*, and the slaves of that group as *coordinator slaves*. The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).

2.2 Directories and Placement

On top of the bag of key-value mappings, the Spanner implementation supports a bucketing abstraction called a *directory*, which is a set of contiguous keys that share a common prefix. (The choice of the term *directory* is a historical accident; a better term might be *bucket*.) We will explain the source of that prefix in Section 2.3. Supporting directories allows applications to control the locality of their data by choosing keys carefully.

A directory is the unit of data placement. All data in a directory has the same replication configuration. When data is moved between Paxos groups, it is moved directory by directory, as shown in Figure 3. Spanner might move a directory to shed load from a Paxos group; to put directories that are frequently accessed together into the same group; or to move a directory into a group that is closer to its accessors. Directories can be moved while client operations are ongoing. One could expect that a 50MB directory can be moved in a few seconds.

The fact that a Paxos group may contain multiple directories implies that a Spanner tablet is different from

a Bigtable tablet: the former is not necessarily a single lexicographically contiguous partition of the row space. Instead, a Spanner tablet is a container that may encapsulate multiple partitions of the row space. We made this decision so that it would be possible to colocate multiple directories that are frequently accessed together.

Movedir is the background task used to move directories between Paxos groups [14]. Movedir is also used to add or remove replicas to Paxos groups [25], because Spanner does not yet support in-Paxos configuration changes. Movedir is not implemented as a single transaction, so as to avoid blocking ongoing reads and writes on a bulky data move. Instead, movedir registers the fact that it is starting to move data and moves the data in the background. When it has moved all but a nominal amount of the data, it uses a transaction to atomically move that nominal amount and update the metadata for the two Paxos groups.

A directory is also the smallest unit whose geographic-replication properties (or *placement*, for short) can be specified by an application. The design of our placement-specification language separates responsibilities for managing replication configurations. Administrators control two dimensions: the number and types of replicas, and the geographic placement of those replicas. They create a menu of named options in these two dimensions (e.g., *North America, replicated 5 ways with 1 witness*). An application controls how data is replicated, by tagging each database and/or individual directories with a combination of those options. For example, an application might store each end-user's data in its own directory, which would enable user *A*'s data to have three replicas in Europe, and user *B*'s data to have five replicas in North America.

For expository clarity we have over-simplified. In fact, Spanner will shard a directory into multiple *fragments* if it grows too large. Fragments may be served from different Paxos groups (and therefore different servers). Movedir actually moves fragments, and not whole directories, between groups.

2.3 Data Model

Spanner exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general-purpose transactions. The move towards supporting these features was driven by many factors. The need to support schematized semi-relational tables and synchronous replication is supported by the popularity of Megastore [5]. At least 300 applications within Google use Megastore (despite its relatively low performance) because its data model is simpler to manage than Bigtable's, and because of its support for synchronous replication across datacenters. (Bigtable only supports eventually-consistent replication across datacenters.) Examples of well-known Google applications that use Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine. The need to support a SQL-like query language in Spanner was also clear, given the popularity of Dremel [28] as an interactive data-analysis tool. Finally, the lack of cross-row transactions in Bigtable led to frequent complaints; Percolator [32] was in part built to address this failing. Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings [9, 10, 19]. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.

The application data model is layered on top of the directory-bucketed key-value mappings supported by the implementation. An application creates one or more *databases* in a universe. Each database can contain an unlimited number of schematized *tables*. Tables look like relational-database tables, with rows, columns, and versioned values. We will not go into detail about the query language for Spanner. It looks like SQL with some extensions to support protocol-buffer-valued fields.

Spanner's data model is not purely relational, in that rows must have names. More precisely, every table is required to have an ordered set of one or more primary-key columns. This requirement is where Spanner still looks like a key-value store: the primary keys form the name for a row, and each table defines a mapping from the primary-key columns to the non-primary-key columns. A row has existence only if some value (even if it is NULL) is defined for the row's keys. Imposing this structure is useful because it lets applications control data locality through their choices of keys.

Figure 4 contains an example Spanner schema for storing photo metadata on a per-user, per-album basis. The schema language is similar to Megastore's, with the additional requirement that every Spanner database must be partitioned by clients into one or more hierarchies of tables. Client applications declare the hierarchies in database schemas via the `INTERLEAVE IN` declarations. The table at the top of a hierarchy is a *directory table*. Each row in a directory table with key *K*, together with all of the rows in descendant tables that start with *K* in lexicographic order, forms a directory. `ON DELETE CASCADE` says that deleting a row in the directory table deletes any associated child rows. The figure also illustrates the interleaved layout for the example database: for

底层的存储结构

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

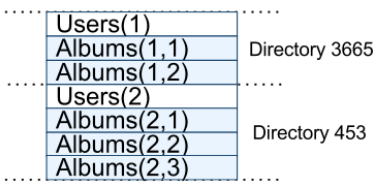

Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

example, `Albums(2,1)` represents the row from the `Albums` table for `user_id 2, album_id 1`. This interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables, which is necessary for good performance in a sharded, distributed database. Without it, Spanner would not know the most important locality relationships.

3 TrueTime

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

Table 1: TrueTime API. The argument t is of type *TTstamp*.

This section describes the TrueTime API and sketches its implementation. We leave most of the details for another paper: our goal is to demonstrate the power of having such an API. Table 1 lists the methods of the API. TrueTime explicitly represents time as a *TTinterval*, which is an interval with bounded time uncertainty (unlike standard time interfaces that give clients no notion of uncertainty). The endpoints of a *TTinterval* are of type *TTstamp*. The *TT.now*() method returns a *TTinterval* that is guaranteed to contain the absolute time during which *TT.now*() was invoked. The time epoch is analogous to UNIX time with leap-second smearing. Define the instantaneous error bound as ϵ, which is half of the interval's width, and the average error bound as $\bar{\epsilon}$. The *TT.after*() and *TT.before*() methods are convenience wrappers around *TT.now*().

Denote the absolute time of an event e by the function $t_{abs}(e)$. In more formal terms, TrueTime guarantees that for an invocation $tt = TT.now()$, $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$, where e_{now} is the invocation event.

The underlying time references used by TrueTime are GPS and atomic clocks. TrueTime uses two forms of time reference because they have different failure modes. GPS reference-source vulnerabilities include antenna and receiver failures, local radio interference, correlated failures (e.g., design faults such as incorrect leap-second handling and spoofing), and GPS system outages. Atomic clocks can fail in ways uncorrelated to GPS and each other, and over long periods of time can drift significantly due to frequency error.

TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine. The majority of masters have GPS receivers with dedicated antennas; these masters are separated physically to reduce the effects of antenna failures, radio interference, and spoofing. The remaining masters (which we refer to as *Armageddon masters*) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master. All masters' time references are regularly compared against each other. Each master also cross-checks the rate at which its reference advances time against its own local clock, and evicts itself if there is substantial divergence. Between synchronizations, Armageddon masters advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift. GPS masters advertise uncertainty that is typically close to zero.

Every daemon polls a variety of masters [29] to reduce vulnerability to errors from any one master. Some are GPS masters chosen from nearby datacenters; the rest are GPS masters from farther datacenters, as well as some Armageddon masters. Daemons apply a variant of Marzullo's algorithm [27] to detect and reject liars, and synchronize the local machine clocks to the non-liars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted.

Between synchronizations, a daemon advertises a slowly increasing time uncertainty. ϵ is derived from conservatively applied worst-case local clock drift. ϵ also depends on time-master uncertainty and communication delay to the time masters. In our production environment, ϵ is typically a sawtooth function of time, varying from about 1 to 7 ms over each poll interval. $\bar{\epsilon}$ is therefore 4 ms most of the time. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at 200 microseconds/second, which together account

Operation	Timestamp Discussion	Concurrency Control	Replica Required
Read-Write Transaction	§ 4.1.2	pessimistic	leader
Read-Only Transaction	§ 4.1.4	lock-free	leader for timestamp; any for read, subject to § 4.1.3
Snapshot Read, client-provided timestamp	—	lock-free	any, subject to § 4.1.3
Snapshot Read, client-provided bound	§ 4.1.3	lock-free	any, subject to § 4.1.3

Table 2: Types of reads and writes in Spanner, and how they compare.

for the sawtooth bounds from 0 to 6 ms. The remaining 1 ms comes from the communication delay to the time masters. Excursions from this sawtooth are possible in the presence of failures. For example, occasional time-master unavailability can cause datacenter-wide increases in ϵ. Similarly, overloaded machines and network links can result in occasional localized ϵ spikes.

4 Concurrency Control

This section describes how TrueTime is used to guarantee the correctness properties around concurrency control, and how those properties are used to implement features such as externally consistent transactions, lock-free read-only transactions, and non-blocking reads in the past. These features enable, for example, the guarantee that a whole-database audit read at a timestamp t will see exactly the effects of every transaction that has committed as of t.

Going forward, it will be important to distinguish writes as seen by Paxos (which we will refer to as *Paxos writes* unless the context is clear) from Spanner client writes. For example, two-phase commit generates a Paxos write for the prepare phase that has no corresponding Spanner client write.

4.1 Timestamp Management

Table 2 lists the types of operations that Spanner supports. The Spanner implementation supports *read-write transactions*, *read-only transactions* (predeclared snapshot-isolation transactions), and *snapshot reads*. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as read-only transactions. Both are internally retried (clients need not write their own retry loops).

A read-only transaction is a kind of transaction that has the performance benefits of snapshot isolation [6]. A read-only transaction must be predeclared as not having any writes; it is not simply a read-write transaction without any writes. Reads in a read-only transaction execute at a system-chosen timestamp without locking, so that incoming writes are not blocked. The execution of the reads in a read-only transaction can proceed on any replica that is sufficiently up-to-date (Section 4.1.3).

A snapshot read is a read in the past that executes without locking. A client can either specify a timestamp for a snapshot read, or provide an upper bound on the desired timestamp's staleness and let Spanner choose a timestamp. In either case, the execution of a snapshot read proceeds at any replica that is sufficiently up-to-date.

For both read-only transactions and snapshot reads, commit is inevitable once a timestamp has been chosen, unless the data at that timestamp has been garbage-collected. As a result, clients can avoid buffering results inside a retry loop. When a server fails, clients can internally continue the query on a different server by repeating the timestamp and the current read position.

4.1.1 Paxos Leader Leases

Spanner's Paxos implementation uses timed leases to make leadership long-lived (10 seconds by default). A potential leader sends requests for timed *lease votes*; upon receiving a quorum of lease votes the leader knows it has a lease. A replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration. Define a leader's *lease interval* as starting when it discovers it has a quorum of lease votes, and as ending when it no longer has a quorum of lease votes (because some have expired). Spanner depends on the following disjointness invariant: for each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's. Appendix A describes how this invariant is enforced.

The Spanner implementation permits a Paxos leader to abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner constrains when abdication is permissible. Define s_{max} to be the maximum timestamp used by a leader. Subsequent sections will describe when s_{max} is advanced. Before abdicating, a leader must wait until $TT.after(s_{max})$ is true.

4.1.2 Assigning Timestamps to RW Transactions

Transactional reads and writes use two-phase locking. As a result, they can be assigned timestamps at any time

when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit.

Spanner depends on the following monotonicity invariant: within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders. A single leader replica can trivially assign timestamps in monotonically increasing order. This invariant is enforced across leaders by making use of the disjointness invariant: a leader must only assign timestamps within the interval of its leader lease. Note that whenever a timestamp s is assigned, s_{max} is advanced to s to preserve disjointness.

Spanner also enforces the following external-consistency invariant: if the start of a transaction T_2 occurs after the commit of a transaction T_1, then the commit timestamp of T_2 must be greater than the commit timestamp of T_1. Define the start and commit events for a transaction T_i by e_i^{start} and e_i^{commit}; and the commit timestamp of a transaction T_i by s_i. The invariant becomes $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$. The protocol for executing transactions and assigning timestamps obeys two rules, which together guarantee this invariant, as shown below. Define the arrival event of the commit request at the coordinator leader for a write T_i to be e_i^{server}.

Start The coordinator leader for a write T_i assigns a commit timestamp s_i no less than the value of $TT.now().latest$, computed after e_i^{server}. Note that the participant leaders do not matter here; Section 4.2.1 describes how they are involved in the implementation of the next rule.

Commit Wait The coordinator leader ensures that clients cannot see any data committed by T_i until $TT.after(s_i)$ is true. Commit wait ensures that s_i is less than the absolute commit time of T_i, or $s_i < t_{abs}(e_i^{commit})$. The implementation of commit wait is described in Section 4.2.1. Proof:

$$
\begin{aligned}
s_1 &< t_{abs}(e_1^{commit}) && \text{(commit wait)} \\
t_{abs}(e_1^{commit}) &< t_{abs}(e_2^{start}) && \text{(assumption)} \\
t_{abs}(e_2^{start}) &\le t_{abs}(e_2^{server}) && \text{(causality)} \\
t_{abs}(e_2^{server}) &\le s_2 && \text{(start)} \\
s_1 &< s_2 && \text{(transitivity)}
\end{aligned}
$$

4.1.3 Serving Reads at a Timestamp

The monotonicity invariant described in Section 4.1.2 allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read. Every replica tracks a value called *safe time* t_{safe} which is the maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp t if $t <= t_{safe}$.

Define $t_{safe} = min(t_{safe}^{Paxos}, t_{safe}^{TM})$, where each Paxos state machine has a safe time t_{safe}^{Paxos} and each transaction manager has a safe time t_{safe}^{TM}. t_{safe}^{Paxos} is simpler: it is the timestamp of the highest-applied Paxos write. Because timestamps increase monotonically and writes are applied in order, writes will no longer occur at or below t_{safe}^{Paxos} with respect to Paxos.

t_{safe}^{TM} is ∞ at a replica if there are zero prepared (but not committed) transactions—that is, transactions in between the two phases of two-phase commit. (For a participant slave, t_{safe}^{TM} actually refers to the replica's leader's transaction manager, whose state the slave can infer through metadata passed on Paxos writes.) If there are any such transactions, then the state affected by those transactions is indeterminate: a participant replica does not know yet whether such transactions will commit. As we discuss in Section 4.2.1, the commit protocol ensures that every participant knows a lower bound on a prepared transaction's timestamp. Every participant leader (for a group g) for a transaction T_i assigns a prepare timestamp $s_{i,g}^{prepare}$ to its prepare record. The coordinator leader ensures that the transaction's commit timestamp $s_i >= s_{i,g}^{prepare}$ over all participant groups g. Therefore, for every replica in a group g, over all transactions T_i prepared at g, $t_{safe}^{TM} = min_i(s_{i,g}^{prepare}) - 1$ over all transactions prepared at g.

4.1.4 Assigning Timestamps to RO Transactions

A read-only transaction executes in two phases: assign a timestamp s_{read} [8], and then execute the transaction's reads as snapshot reads at s_{read}. The snapshot reads can execute at any replicas that are sufficiently up-to-date.

The simple assignment of $s_{read} = TT.now().latest$, at any time after a transaction starts, preserves external consistency by an argument analogous to that presented for writes in Section 4.1.2. However, such a timestamp may require the execution of the data reads at s_{read} to block if t_{safe} has not advanced sufficiently. (In addition, note that choosing a value of s_{read} may also advance s_{max} to preserve disjointness.) To reduce the chances of blocking, Spanner should assign the oldest timestamp that preserves external consistency. Section 4.2.2 explains how such a timestamp can be chosen.

4.2 Details

This section explains some of the practical details of read-write transactions and read-only transactions elided earlier, as well as the implementation of a special transaction type used to implement atomic schema changes.

It then describes some refinements of the basic schemes as described.

4.2.1 Read-Write Transactions

Like Bigtable, writes that occur in a transaction are buffered at the client until commit. As a result, reads in a transaction do not see the effects of the transaction's writes. This design works well in Spanner because a read returns the timestamps of any data read, and uncommitted writes have not yet been assigned timestamps.

Reads within read-write transactions use wound-wait [33] to avoid deadlocks. The client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data. While a client transaction remains open, it sends keepalive messages to prevent participant leaders from timing out its transaction. When a client has completed all reads and buffered all writes, it begins two-phase commit. The client chooses a coordinator group and sends a commit message to each participant's leader with the identity of the coordinator and any buffered writes. Having the client drive two-phase commit avoids sending data twice across wide-area links.

A non-coordinator-participant leader first acquires write locks. It then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions (to preserve monotonicity), and logs a prepare record through Paxos. Each participant then notifies the coordinator of its prepare timestamp.

The coordinator leader also first acquires write locks, but skips the prepare phase. It chooses a timestamp for the entire transaction after hearing from all other participant leaders. The commit timestamp s must be greater or equal to all prepare timestamps (to satisfy the constraints discussed in Section 4.1.3), greater than $TT.now().latest$ at the time the coordinator received its commit message, and greater than any timestamps the leader has assigned to previous transactions (again, to preserve monotonicity). The coordinator leader then logs a commit record through Paxos (or an abort if it timed out while waiting on the other participants).

Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until $TT.after(s)$, so as to obey the commit-wait rule described in Section 4.1.2. Because the coordinator leader chose s based on $TT.now().latest$, and now waits until that timestamp is guaranteed to be in the past, the expected wait is at least $2 * \bar{\epsilon}$. This wait is typically overlapped with Paxos communication. After commit wait, the coordinator sends the commit timestamp to the client and all other participant leaders. Each participant leader logs the transaction's outcome through Paxos. All participants apply at the same timestamp and then release locks.

非协调者leader，而是参与者leader

4.2.2 Read-Only Transactions

Assigning a timestamp requires a negotiation phase between all of the Paxos groups that are involved in the reads. As a result, Spanner requires a *scope* expression for every read-only transaction, which is an expression that summarizes the keys that will be read by the entire transaction. Spanner automatically infers the scope for standalone queries.

If the scope's values are served by a single Paxos group, then the client issues the read-only transaction to that group's leader. (The current Spanner implementation only chooses a timestamp for a read-only transaction at a Paxos leader.) That leader assigns s_{read} and executes the read. For a single-site read, Spanner generally does better than $TT.now().latest$. Define $LastTS()$ to be the timestamp of the last committed write at a Paxos group. If there are no prepared transactions, the assignment $s_{read} = LastTS()$ trivially satisfies external consistency: the transaction will see the result of the last write, and therefore be ordered after it.

If the scope's values are served by multiple Paxos groups, there are several options. The most complicated option is to do a round of communication with all of the groups's leaders to negotiate s_{read} based on $LastTS()$. Spanner currently implements a simpler choice. The client avoids a negotiation round, and just has its reads execute at $s_{read} = TT.now().latest$ (which may wait for safe time to advance). All reads in the transaction can be sent to replicas that are sufficiently up-to-date.

4.2.3 Schema-Change Transactions

TrueTime enables Spanner to support atomic schema changes. It would be infeasible to use a standard transaction, because the number of participants (the number of groups in a database) could be in the millions. Bigtable supports atomic schema changes in one datacenter, but its schema changes block all operations.

A Spanner schema-change transaction is a generally non-blocking variant of a standard transaction. First, it is explicitly assigned a timestamp in the future, which is registered in the prepare phase. As a result, schema changes across thousands of servers can complete with minimal disruption to other concurrent activity. Second, reads and writes, which implicitly depend on the schema, synchronize with any registered schema-change timestamp at time t: they may proceed if their timestamps precede t, but they must block behind the schema-change transaction if their timestamps are after t. Without TrueTime, defining the schema change to happen at t would be meaningless.

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

4.2.4 Refinements

t_{safe}^{TM} as defined above has a weakness, in that a single prepared transaction prevents t_{safe} from advancing. As a result, no reads can occur at later timestamps, even if the reads do not conflict with the transaction. Such false conflicts can be removed by augmenting t_{safe}^{TM} with a fine-grained mapping from key ranges to prepared-transaction timestamps. This information can be stored in the lock table, which already maps key ranges to lock metadata. When a read arrives, it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts.

LastTS() as defined above has a similar weakness: if a transaction has just committed, a non-conflicting read-only transaction must still be assigned s_{read} so as to follow that transaction. As a result, the execution of the read could be delayed. This weakness can be remedied similarly by augmenting *LastTS*() with a fine-grained mapping from key ranges to commit timestamps in the lock table. (We have not yet implemented this optimization.) When a read-only transaction arrives, its timestamp can be assigned by taking the maximum value of *LastTS*() for the key ranges with which the transaction conflicts, unless there is a conflicting prepared transaction (which can be determined from fine-grained safe time).

t_{safe}^{Paxos} as defined above has a weakness in that it cannot advance in the absence of Paxos writes. That is, a snapshot read at t cannot execute at Paxos groups whose last write happened before t. Spanner addresses this problem by taking advantage of the disjointness of leader-lease intervals. Each Paxos leader advances t_{safe}^{Paxos} by keeping a threshold above which future writes' timestamps will occur: it maintains a mapping *MinNextTS*(n) from Paxos sequence number n to the minimum timestamp that may be assigned to Paxos sequence number $n+1$. A replica can advance t_{safe}^{Paxos} to *MinNextTS*$(n) - 1$ when it has applied through n.

A single leader can enforce its *MinNextTS*() promises easily. Because the timestamps promised by *MinNextTS*() lie within a leader's lease, the disjointness invariant enforces *MinNextTS*() promises across leaders. If a leader wishes to advance *MinNextTS*() beyond the end of its leader lease, it must first extend its lease. Note that s_{max} is always advanced to the highest value in *MinNextTS*() to preserve disjointness.

A leader by default advances *MinNextTS*() values every 8 seconds. Thus, in the absence of prepared transactions, healthy slaves in an idle Paxos group can serve reads at timestamps greater than 8 seconds old in the worst case. A leader may also advance *MinNextTS*() values on demand from slaves.

5 Evaluation

We first measure Spanner's performance with respect to replication, transactions, and availability. We then provide some data on TrueTime behavior, and a case study of our first client, F1.

5.1 Microbenchmarks

Table 3 presents some microbenchmarks for Spanner. These measurements were taken on timeshared machines: each spanserver ran on scheduling units of 4GB RAM and 4 cores (AMD Barcelona 2200MHz). Clients were run on separate machines. Each zone contained one spanserver. Clients and zones were placed in a set of datacenters with network distance of less than 1ms. (Such a layout should be commonplace: most applications do not need to distribute all of their data worldwide.) The test database was created with 50 Paxos groups with 2500 directories. Operations were standalone reads and writes of 4KB. All reads were served out of memory after a compaction, so that we are only measuring the overhead of Spanner's call stack. In addition, one unmeasured round of reads was done first to warm any location caches.

系统benchmark方法

For the latency experiments, clients issued sufficiently few operations so as to avoid queuing at the servers. From the 1-replica experiments, commit wait is about 5ms, and Paxos latency is about 9ms. As the number of replicas increases, the latency stays roughly constant with less standard deviation because Paxos executes in parallel at a group's replicas. As the number of replicas increases, the latency to achieve a quorum becomes less sensitive to slowness at one slave replica.

For the throughput experiments, clients issued sufficiently many operations so as to saturate the servers'

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

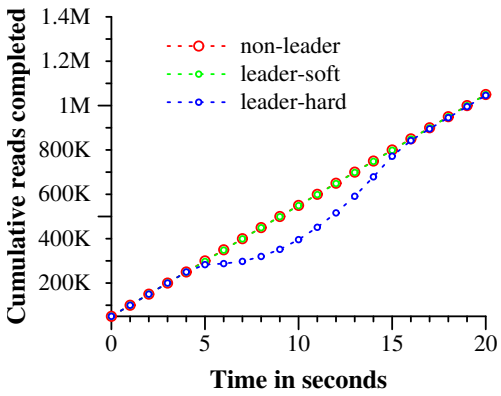

Figure 5: Effect of killing servers on throughput.

CPUs. Snapshot reads can execute at any up-to-date replicas, so their throughput increases almost linearly with the number of replicas. Single-read read-only transactions only execute at leaders because timestamp assignment must happen at leaders. Read-only-transaction throughput increases with the number of replicas because the number of effective spanservers increases: in the experimental setup, the number of spanservers equaled the number of replicas, and leaders were randomly distributed among the zones. Write throughput benefits from the same experimental artifact (which explains the increase in throughput from 3 to 5 replicas), but that benefit is outweighed by the linear increase in the amount of work performed per write, as the number of replicas increases.

Table 4 demonstrates that two-phase commit can scale to a reasonable number of participants: it summarizes a set of experiments run across 3 zones, each with 25 spanservers. Scaling up to 50 participants is reasonable in both mean and 99th-percentile, and latencies start to rise noticeably at 100 participants.

5.2 Availability

Figure 5 illustrates the availability benefits of running Spanner in multiple datacenters. It shows the results of three experiments on throughput in the presence of datacenter failure, all of which are overlaid onto the same time scale. The test universe consisted of 5 zones Z_i, each of which had 25 spanservers. The test database was sharded into 1250 Paxos groups, and 100 test clients constantly issued non-snapshot reads at an aggregrate rate of 50K reads/second. All of the leaders were explicitly placed in Z_1. Five seconds into each test, all of the servers in one zone were killed: *non-leader* kills Z_2; *leader-hard* kills Z_1; *leader-soft* kills Z_1, but it gives notifications to all of the servers that they should handoff leadership first.

Killing Z_2 has no effect on read throughput. Killing Z_1 while giving the leaders time to handoff leadership to a different zone has a minor effect: the throughput drop is not visible in the graph, but is around 3-4%. On the other hand, killing Z_1 with no warning has a severe effect: the rate of completion drops almost to 0. As leaders get re-elected, though, the throughput of the system rises to approximately 100K reads/second because of two artifacts of our experiment: there is extra capacity in the system, and operations are queued while the leader is unavailable. As a result, the throughput of the system rises before leveling off again at its steady-state rate.

We can also see the effect of the fact that Paxos leader leases are set to 10 seconds. When we kill the zone, the leader-lease expiration times for the groups should be evenly distributed over the next 10 seconds. Soon after each lease from a dead leader expires, a new leader is elected. Approximately 10 seconds after the kill time, all of the groups have leaders and throughput has recovered. Shorter lease times would reduce the effect of server deaths on availability, but would require greater amounts of lease-renewal network traffic. We are in the process of designing and implementing a mechanism that will cause slaves to release Paxos leader leases upon leader failure.

5.3 TrueTime

Two questions must be answered with respect to TrueTime: is ϵ truly a bound on clock uncertainty, and how bad does ϵ get? For the former, the most serious problem would be if a local clock's drift were greater than 200us/sec: that would break assumptions made by TrueTime. Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems. As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends.

Figure 6 presents TrueTime data taken at several thousand spanserver machines across datacenters up to 2200

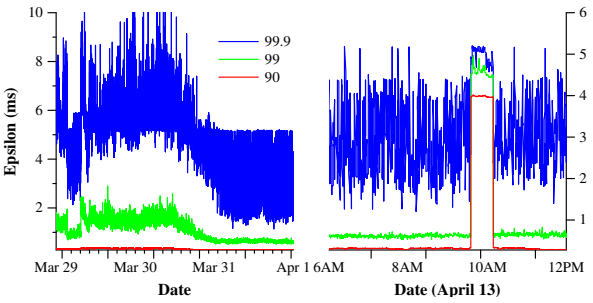

Figure 6: Distribution of TrueTime ϵ values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

km apart. It plots the 90th, 99th, and 99.9th percentiles of ϵ, sampled at timeslave daemons immediately after polling the time masters. This sampling elides the sawtooth in ϵ due to local-clock uncertainty, and therefore measures time-master uncertainty (which is generally 0) plus communication delay to the time masters.

The data shows that these two factors in determining the base value of ϵ are generally not a problem. However, there can be significant tail-latency issues that cause higher values of ϵ. The reduction in tail latencies beginning on March 30 were due to networking improvements that reduced transient network-link congestion. The increase in ϵ on April 13, approximately one hour in duration, resulted from the shutdown of 2 time masters at a datacenter for routine maintenance. We continue to investigate and remove causes of TrueTime spikes.

5.4 F1

Spanner started being experimentally evaluated under production workloads in early 2011, as part of a rewrite of Google's advertising backend called F1 [35]. This backend was originally based on a MySQL database that was manually sharded many ways. The uncompressed dataset is tens of terabytes, which is small compared to many NoSQL instances, but was large enough to cause difficulties with sharded MySQL. The MySQL sharding scheme assigned each customer and all related data to a fixed shard. This layout enabled the use of indexes and complex query processing on a per-customer basis, but required some knowledge of the sharding in application business logic. Resharding this revenue-critical database as it grew in the number of customers and their data was extremely costly. The last resharding took over two years of intense effort, and involved coordination and testing across dozens of teams to minimize risk. This operation was too complex to do regularly: as a result, the team had to limit growth on the MySQL database by storing some data in external Bigtables, which compromised transactional behavior and the ability to query across all data.

# fragments	# directories
1	>100M
2–4	341
5–9	5336
10–14	232
15–99	34
100–500	7

Table 5: Distribution of directory-fragment counts in F1.

The F1 team chose to use Spanner for several reasons. First, Spanner removes the need to manually reshard. Second, Spanner provides synchronous replication and automatic failover. With MySQL master-slave replication, failover was difficult, and risked data loss and downtime. Third, F1 requires strong transactional semantics, which made using other NoSQL systems impractical. Application semantics requires transactions across arbitrary data, and consistent reads. The F1 team also needed secondary indexes on their data (since Spanner does not yet provide automatic support for secondary indexes), and was able to implement their own consistent global indexes using Spanner transactions.

All application writes are now by default sent through F1 to Spanner, instead of the MySQL-based application stack. F1 has 2 replicas on the west coast of the US, and 3 on the east coast. This choice of replica sites was made to cope with outages due to potential major natural disasters, and also the choice of their frontend sites. Anecdotally, Spanner's automatic failover has been nearly invisible to them. Although there have been unplanned cluster failures in the last few months, the most that the F1 team has had to do is update their database's schema to tell Spanner where to preferentially place Paxos leaders, so as to keep them close to where their frontends moved.

Spanner's timestamp semantics made it efficient for F1 to maintain in-memory data structures computed from the database state. F1 maintains a logical history log of all changes, which is written into Spanner itself as part of every transaction. F1 takes full snapshots of data at a timestamp to initialize its data structures, and then reads incremental changes to update them.

Table 5 illustrates the distribution of the number of fragments per directory in F1. Each directory typically corresponds to a customer in the application stack above F1. The vast majority of directories (and therefore customers) consist of only 1 fragment, which means that reads and writes to those customers' data are guaranteed to occur on only a single server. The directories with more than 100 fragments are all tables that contain F1 secondary indexes: writes to more than a few fragments

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

of such tables are extremely uncommon. The F1 team has only seen such behavior when they do untuned bulk data loads as transactions.

Table 6 presents Spanner operation latencies as measured from F1 servers. Replicas in the east-coast data centers are given higher priority in choosing Paxos leaders. The data in the table is measured from F1 servers in those data centers. The large standard deviation in write latencies is caused by a pretty fat tail due to lock conflicts. The even larger standard deviation in read latencies is partially due to the fact that Paxos leaders are spread across two data centers, only one of which has machines with SSDs. In addition, the measurement includes every read in the system from two datacenters: the mean and standard deviation of the bytes read were roughly 1.6KB and 119KB, respectively.

6 Related Work

Consistent replication across datacenters as a storage service has been provided by Megastore [5] and DynamoDB [3]. DynamoDB presents a key-value interface, and only replicates within a region. Spanner follows Megastore in providing a semi-relational data model, and even a similar schema language. Megastore does not achieve high performance. It is layered on top of Bigtable, which imposes high communication costs. It also does not support long-lived leaders: multiple replicas may initiate writes. All writes from different replicas necessarily conflict in the Paxos protocol, even if they do not logically conflict: throughput collapses on a Paxos group at several writes per second. Spanner provides higher performance, general-purpose transactions, and external consistency.

Pavlo et al. [31] have compared the performance of databases and MapReduce [12]. They point to several other efforts that have been made to explore database functionality layered on distributed key-value stores [1, 4, 7, 41] as evidence that the two worlds are converging. We agree with the conclusion, but demonstrate that integrating multiple layers has its advantages: integrating concurrency control with replication reduces the cost of commit wait in Spanner, for example.

The notion of layering transactions on top of a replicated store dates at least as far back as Gifford's dissertation [16]. Scatter [17] is a recent DHT-based key-value store that layers transactions on top of consistent replication. Spanner focuses on providing a higher-level interface than Scatter does. Gray and Lamport [18] describe a non-blocking commit protocol based on Paxos. Their protocol incurs more messaging costs than two-phase commit, which would aggravate the cost of commit over widely distributed groups. Walter [36] provides a variant of snapshot isolation that works within, but not across datacenters. In contrast, our read-only transactions provide a more natural semantics, because we support external consistency over all operations.

There has been a spate of recent work on reducing or eliminating locking overheads. Calvin [40] eliminates concurrency control: it pre-assigns timestamps and then executes the transactions in timestamp order. H-Store [39] and Granola [11] each supported their own classification of transaction types, some of which could avoid locking. None of these systems provides external consistency. Spanner addresses the contention issue by providing support for snapshot isolation.

VoltDB [42] is a sharded in-memory database that supports master-slave replication over the wide area for disaster recovery, but not more general replication configurations. It is an example of what has been called NewSQL, which is a marketplace push to support scalable SQL [38]. A number of commercial databases implement reads in the past, such as MarkLogic [26] and Oracle's Total Recall [30]. Lomet and Li [24] describe an implementation strategy for such a temporal database.

Farsite derived bounds on clock uncertainty (much looser than TrueTime's) relative to a trusted clock reference [13]: server leases in Farsite were maintained in the same way that Spanner maintains Paxos leases. Loosely synchronized clocks have been used for concurrency-control purposes in prior work [2, 23]. We have shown that TrueTime lets one reason about global time across sets of Paxos state machines.

7 Future Work

We have spent most of the last year working with the F1 team to transition Google's advertising backend from MySQL to Spanner. We are actively improving its monitoring and support tools, as well as tuning its performance. In addition, we have been working on improving the functionality and performance of our backup/restore system. We are currently implementing the Spanner schema language, automatic maintenance of secondary indices, and automatic load-based resharding. Longer term, there are a couple of features that we plan to in-

vestigate. Optimistically doing reads in parallel may be a valuable strategy to pursue, but initial experiments have indicated that the right implementation is non-trivial. In addition, we plan to eventually support direct changes of Paxos configurations [22, 34].

Given that we expect many applications to replicate their data across datacenters that are relatively close to each other, TrueTime ϵ may noticeably affect performance. We see no insurmountable obstacle to reducing ϵ below 1ms. Time-master-query intervals can be reduced, and better clock crystals are relatively cheap. Time-master query latency could be reduced with improved networking technology, or possibly even avoided through alternate time-distribution technology.

Finally, there are obvious areas for improvement. Although Spanner is scalable in the number of nodes, the node-local data structures have relatively poor performance on complex SQL queries, because they were designed for simple key-value accesses. Algorithms and data structures from DB literature could improve single-node performance a great deal. Second, moving data automatically between datacenters in response to changes in client load has long been a goal of ours, but to make that goal effective, we would also need the ability to move client-application processes between datacenters in an automated, coordinated fashion. Moving processes raises the even more difficult problem of managing resource acquisition and allocation between datacenters.

8 Conclusions

To summarize, Spanner combines and extends on ideas from two research communities: from the database community, a familiar, easy-to-use, semi-relational interface, transactions, and an SQL-based query language; from the systems community, scalability, automatic sharding, fault tolerance, consistent replication, external consistency, and wide-area distribution. Since Spanner's inception, we have taken more than 5 years to iterate to the current design and implementation. Part of this long iteration phase was due to a slow realization that Spanner should do more than tackle the problem of a globally-replicated namespace, and should also focus on database features that Bigtable was missing.

One aspect of our design stands out: the linchpin of Spanner's feature set is TrueTime. We have shown that reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics. In addition, as the underlying system enforces tighter bounds on clock uncertainty, the overhead of the stronger semantics decreases. As a community, we should no longer depend on loosely synchronized clocks and weak time APIs in designing distributed algorithms.

Acknowledgements

Many people have helped to improve this paper: our shepherd Jon Howell, who went above and beyond his responsibilities; the anonymous referees; and many Googlers: Atul Adya, Fay Chang, Frank Dabek, Sean Dorward, Bob Gruber, David Held, Nick Kline, Alex Thomson, and Joel Wein. Our management has been very supportive of both our work and of publishing this paper: Aristotle Balogh, Bill Coughran, Urs Hölzle, Doron Meyer, Cos Nicolaou, Kathy Polizzi, Sridhar Ramaswany, and Shivakumar Venkataraman.

We have built upon the work of the Bigtable and Megastore teams. The F1 team, and Jeff Shute in particular, worked closely with us in developing our data model and helped immensely in tracking down performance and correctness bugs. The Platforms team, and Luiz Barroso and Bob Felderman in particular, helped to make TrueTime happen. Finally, a lot of Googlers used to be on our team: Ken Ashcraft, Paul Cychosz, Krzysztof Ostrowski, Amir Voskoboynik, Matthew Weaver, Theo Vassilakis, and Eric Veach; or have joined our team recently: Nathan Bales, Adam Beberg, Vadim Borisov, Ken Chen, Brian Cooper, Cian Cullinan, Robert-Jan Huijsman, Milind Joshi, Andrey Khorlin, Dawid Kuroczko, Laramie Leavitt, Eric Li, Mike Mammarella, Sunil Mushran, Simon Nielsen, Ovidiu Platon, Ananth Shrinivas, Vadim Suvorov, and Marcel van der Holst.

References

[1] Azza Abouzeid et al. "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads". *Proc. of VLDB*. 2009, pp. 922–933.

[2] A. Adya et al. "Efficient optimistic concurrency control using loosely synchronized clocks". *Proc. of SIGMOD*. 1995, pp. 23–34.

[3] Amazon. *Amazon DynamoDB*. 2012.

[4] Michael Armbrust et al. "PIQL: Success-Tolerant Query Processing in the Cloud". *Proc. of VLDB*. 2011, pp. 181–192.

[5] Jason Baker et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". *Proc. of CIDR*. 2011, pp. 223–234.

[6] Hal Berenson et al. "A critique of ANSI SQL isolation levels". *Proc. of SIGMOD*. 1995, pp. 1–10.

[7] Matthias Brantner et al. "Building a database on S3". *Proc. of SIGMOD*. 2008, pp. 251–264.

[8] A. Chan and R. Gray. "Implementing Distributed Read-Only Transactions". *IEEE TOSE* SE-11.2 (Feb. 1985), pp. 205–212.

[9] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". *ACM TOCS* 26.2 (June 2008), 4:1–4:26.

[10] Brian F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". *Proc. of VLDB*. 2008, pp. 1277–1288.

[11] James Cowling and Barbara Liskov. "Granola: Low-Overhead Distributed Transaction Coordination". *Proc. of USENIX ATC*. 2012, pp. 223–236.

[12] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: a flexible data processing tool". *CACM* 53.1 (Jan. 2010), pp. 72–77.

[13] John Douceur and Jon Howell. *Scalable Byzantine-Fault-Quantifying Clock Synchronization*. Tech. rep. MSR-TR-2003-67. MS Research, 2003.

[14] John R. Douceur and Jon Howell. "Distributed directory service in the Farsite file system". *Proc. of OSDI*. 2006, pp. 321–334.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". *Proc. of SOSP*. Dec. 2003, pp. 29–43.

[16] David K. Gifford. *Information Storage in a Decentralized Computer System*. Tech. rep. CSL-81-8. PhD dissertation. Xerox PARC, July 1982.

[17] Lisa Glendenning et al. "Scalable consistency in Scatter". *Proc. of SOSP*. 2011.

[18] Jim Gray and Leslie Lamport. "Consensus on transaction commit". *ACM TODS* 31.1 (Mar. 2006), pp. 133–160.

[19] Pat Helland. "Life beyond Distributed Transactions: an Apostate's Opinion". *Proc. of CIDR*. 2007, pp. 132–141.

[20] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". *ACM TOPLAS* 12.3 (July 1990), pp. 463–492.

[21] Leslie Lamport. "The part-time parliament". *ACM TOCS* 16.2 (May 1998), pp. 133–169.

[22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. "Reconfiguring a state machine". *SIGACT News* 41.1 (Mar. 2010), pp. 63–73.

[23] Barbara Liskov. "Practical uses of synchronized clocks in distributed systems". *Distrib. Comput.* 6.4 (July 1993), pp. 211–219.

[24] David B. Lomet and Feifei Li. "Improving Transaction-Time DBMS Performance and Functionality". *Proc. of ICDE* (2009), pp. 581–591.

[25] Jacob R. Lorch et al. "The SMART way to migrate replicated stateful services". *Proc. of EuroSys*. 2006, pp. 103–115.

[26] MarkLogic. *MarkLogic 5 Product Documentation*. 2012.

[27] Keith Marzullo and Susan Owicki. "Maintaining the time in a distributed system". *Proc. of PODC*. 1983, pp. 295–305.

[28] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". *Proc. of VLDB*. 2010, pp. 330–339.

[29] D.L. Mills. *Time synchronization in DCNET hosts*. Internet Project Report IEN–173. COMSAT Laboratories, Feb. 1981.

[30] Oracle. *Oracle Total Recall*. 2012.

[31] Andrew Pavlo et al. "A comparison of approaches to large-scale data analysis". *Proc. of SIGMOD*. 2009, pp. 165–178.

[32] Daniel Peng and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications". *Proc. of OSDI*. 2010, pp. 1–15.

[33] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. "System level concurrency control for distributed database systems". *ACM TODS* 3.2 (June 1978), pp. 178–198.

[34] Alexander Shraer et al. "Dynamic Reconfiguration of Primary/Backup Clusters". *Proc. of USENIX ATC*. 2012, pp. 425–438.

[35] Jeff Shute et al. "F1 — The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business". *Proc. of SIGMOD*. May 2012, pp. 777–778.

[36] Yair Sovran et al. "Transactional storage for geo-replicated systems". *Proc. of SOSP*. 2011, pp. 385–400.

[37] Michael Stonebraker. *Why Enterprises Are Uninterested in NoSQL*. 2010.

[38] Michael Stonebraker. *Six SQL Urban Myths*. 2010.

[39] Michael Stonebraker et al. "The end of an architectural era: (it's time for a complete rewrite)". *Proc. of VLDB*. 2007, pp. 1150–1160.

[40] Alexander Thomson et al. "Calvin: Fast Distributed Transactions for Partitioned Database Systems". *Proc. of SIGMOD*. 2012, pp. 1–12.

[41] Ashish Thusoo et al. "Hive — A Petabyte Scale Data Warehouse Using Hadoop". *Proc. of ICDE*. 2010, pp. 996–1005.

[42] VoltDB. *VoltDB Resources*. 2012.

A Paxos Leader-Lease Management

The simplest means to ensure the disjointness of Paxos-leader-lease intervals would be for a leader to issue a synchronous Paxos write of the lease interval, whenever it would be extended. A subsequent leader would read the interval and wait until that interval has passed.

TrueTime can be used to ensure disjointness without these extra log writes. The potential ith leader keeps a lower bound on the start of a lease vote from replica r as $v_{i,r}^{leader} = \mathit{TT.now}().\mathit{earliest}$, computed before $e_{i,r}^{send}$ (defined as when the lease request is sent by the leader). Each replica r grants a lease at lease $e_{i,r}^{grant}$, which happens after $e_{i,r}^{receive}$ (when the replica receives a lease request); the lease ends at $t_{i,r}^{end} = \mathit{TT.now}().\mathit{latest} + 10$, computed after $e_{i,r}^{receive}$. A replica r obeys the **single-vote** rule: it will not grant another lease vote until $\mathit{TT.after}(t_{i,r}^{end})$ is true. To enforce this rule across different incarnations of r, Spanner logs a lease vote at the granting replica before granting the lease; this log write can be piggybacked upon existing Paxos-protocol log writes.

When the ith leader receives a quorum of votes (event e_i^{quorum}), it computes its lease interval as $\mathit{lease}_i = [\mathit{TT.now}().\mathit{latest}, \mathit{min}_r(v_{i,r}^{leader}) + 10]$. The lease is deemed to have expired at the leader when $\mathit{TT.before}(\mathit{min}_r(v_{i,r}^{leader}) + 10)$ is false. To prove disjointness, we make use of the fact that the ith and $(i+1)$th leaders must have one replica in common in their quorums. Call that replica $r0$. Proof:

$$
\begin{aligned}
\mathit{lease}_i.\mathit{end} &= \mathit{min}_r(v_{i,r}^{leader}) + 10 && \text{(by definition)}\\
\mathit{min}_r(v_{i,r}^{leader}) + 10 &\le v_{i,r0}^{leader} + 10 && \text{(min)}\\
v_{i,r0}^{leader} + 10 &\le t_{abs}(e_{i,r0}^{send}) + 10 && \text{(by definition)}\\
t_{abs}(e_{i,r0}^{send}) + 10 &\le t_{abs}(e_{i,r0}^{receive}) + 10 && \text{(causality)}\\
t_{abs}(e_{i,r0}^{receive}) + 10 &\le t_{i,r0}^{end} && \text{(by definition)}\\
t_{i,r0}^{end} &< t_{abs}(e_{i+1,r0}^{grant}) && \text{(single-vote)}\\
t_{abs}(e_{i+1,r0}^{grant}) &\le t_{abs}(e_{i+1}^{quorum}) && \text{(causality)}\\
t_{abs}(e_{i+1}^{quorum}) &\le \mathit{lease}_{i+1}.\mathit{start} && \text{(by definition)}
\end{aligned}
$$