Transparent Migration from Datastore to Firestore

Ed Davisson
Google, Inc.
eddavisson@google.com

Tilo Dickopp
Google, Inc.
tdickopp@google.com

David Gay
Google, Inc.
dgay@google.com

Eric Karasuda
Google, Inc.
karasuda@google.com

Ram Kesavan
Google, Inc.
ram.kesavan@gmail.com

Vadim Yushprakh
Google, Inc.
vadimy@google.com

ABSTRACT

Datastore was one of Google's first cloud databases, launched initially as part of App Engine, and built over Google's internal Megastore database system. Firestore was launched in 2019, both a re-implementation of Datastore over Google's Spanner database system and a new, mobile and web-friendly Firestore API. Spanner was chosen as the storage engine of Firestore in particular for technical reasons—it provides unrestricted transaction capabilities, strong consistency guarantees, and other improvements over Megastore.

To provide these improvements to all our customers, and simplify our overall system, a non-disruptive, zero-downtime migration was executed of all Datastore databases (stored in Megastore) to Firestore databases (stored in Spanner). This migration took a couple of years to design and plan, and about three to execute. This paper describes both the core engine for migrating databases, and various practical problems that were solved to make this journey successful. As of the writing of this paper, all (over one million) databases have been successfully migrated.

PVLDB Reference Format:
Ed Davisson, Tilo Dickopp, David Gay, Eric Karasuda, Ram Kesavan, and Vadim Yushprakh. Transparent Migration from Datastore to Firestore. PVLDB, 17(12): 3960 - 3972, 2024.
doi:10.14778/3685800.3685819

1 INTRODUCTION

Datastore launched in 2008 as the database for Google's early platform-as-a-service App Engine [34]. Megastore [4] was used as Datastore's storage layer. A cloud API, "Cloud Datastore" was added in 2013, making the Datastore service accessible to the entire Google cloud platform. Firestore launched in 2019, combining a newer implementation of the Datastore API with a new mobile and web-friendly Firestore API [18], but used Spanner [10] as its storage layer. The two (Datastore and Firestore) APIs share a common data model and can access the same underlying data; they are also priced identically. However, the new implementation of the Datastore API brought two notable improvements: strong consistency for all queries, and unrestricted transactions.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685819

Two strong reasons compelled us to migrate Datastore databases on Megastore to Firestore on Spanner: the improvements noted clearly make the new (Firestore's) re-implementation of the Datastore API better than the original, and maintaining and improving two implementations was not practical in the long term, especially as Megastore is now legacy technology (neither maintained nor improved). The design and planning took a couple of years, with migration of customer databases starting in 2021. At that point, Datastore had over a million databases, as well as significant storage and QPS.

To ensure the success of this migration, we followed a few principles: (1) Data integrity is preserved during and after migration. (2) Migration causes no downtime or disruption to customer traffic during and after migration—a zero-downtime migration was promised when Firestore was launched [24], and is a vital requirement for our larger customers as downtime can negatively affect their revenue and reputation. (3) No customer database is left behind. (4) We communicate to the customer about their migration, and they must explicitly permit any performance or behavioral regression. (5) The migration is automatic; the customer does not need to take any action. The alternative of a self-administered migration would burden customers, create a substantial communication and support cost, and fail to ensure that no customer database is left behind.

We make the following contributions in this paper:

- We describe the key building blocks of *MegaMover*, the core engine used for the safe and non-disruptive migration of large amounts of data and customer traffic. We show how the migrated data is validated to ensure data integrity.
- We discuss the performance of the Datastore and Firestore storage layers, in particular the differences that follow from different concurrency models, to predict regressions that may impact a customer application.
- We discuss the changes made to Firestore's concurrency control mechanisms in response to performance analysis. These changes move Firestore from being "clearly better" than Datastore to always better (for all customers).
- We present practical topics and insights: the ordering of databases for migration, the automation needed, testing techniques and infrastructure, and the capability to stop and undo migration, which is critical given our inability to predict all problem scenarios a priori.

Although live migration of databases is not a new problem, we believe it has rarely been attempted at this scale (number of databases, total QPS and storage). Our goal in writing this paper is to share our learnings with practitioners who are migrating or attempting to

migrate one or more layers underneath a large number of databases without disrupting existing applications. An extended version of this paper [11] includes additional details, especially in the area of performance.

2 BACKGROUND

Datastore offered a highly available, durable, and schemaless NoSQL database that easily scaled to handle applications' load by automatically sharding and replicating the data. Millions of Datastore (and now Firestore) databases have since been created by customers, including the server side of a popular social media app with many millions of users and a popular billion-dollar mobile game.

Data in a Datastore database is organized as a hierarchy of schemaless *entities*. Each entity has a unique key and named properties with values from a rich type set—integers, strings, dates, booleans, maps, etc. A key is a path-like list of *kind* (a string) and *identifier* (a string or 64-bit integer) pairs. For example:

```
/users/752/rooms/den: // key
  "size": 100,        // integer property
  "furniture":        // map property
    { "desk": true, "bed": false, "chair": true }
```

is an entity with two properties and a key with two pairs. The first one has kind "users" and identifier "752". The last kind in this list is the *entity's kind* ("rooms" in this example), which behaves akin to a database table in Datastore queries. For example,

```
SELECT * FROM rooms ORDER BY size
```

returns all entities from the "rooms" kind ordered by ascending value of their "size" property.

The keys of entities constitute an implicit hierarchy: the entity with key /users/752/rooms/dens is a *child* of the one with key /users/752; the latter is the *parent*. All entities that share the same first (kind, identifier) pair belong to the same *entity group*. For example, the following four entities all belong to the (users, 752) entity group: /users/752, /users/752/rooms/den, /users/752/rooms/kitchen, and /users/752/backups/A/slice/23.

The Datastore API provides capabilities to read and write entities, ACID transactions, and a simple SQL-like query engine with indexes for efficient querying. By default, a Datastore database maintains *built-in indexes* for each property of each entity kind. Additionally, the customer can configure *composite indexes* across multiple properties.

Datastore was built on Megastore, which used Paxos [28] to synchronously replicate entity groups for high availability; each replica was hosted in Bigtable [8]. Megastore, and hence Datastore, provided full ACID semantics within each entity group but no consistency guarantees across them. Furthermore, each entity group had a limited maximum write rate (on the order of a few writes per second), forcing application developers to split their data across many entity groups, and then have to resort to eventually consistent queries to access this data. These consistency limitations significantly complicated application—and in particular, data model—development. As a mitigation, Datastore also supported transactions that straddle up to 25 entity groups [16]. The migration of these databases to Firestore (built over Spanner) results in the complete removal of these write rate and consistency limitations, which is very attractive to our customers.

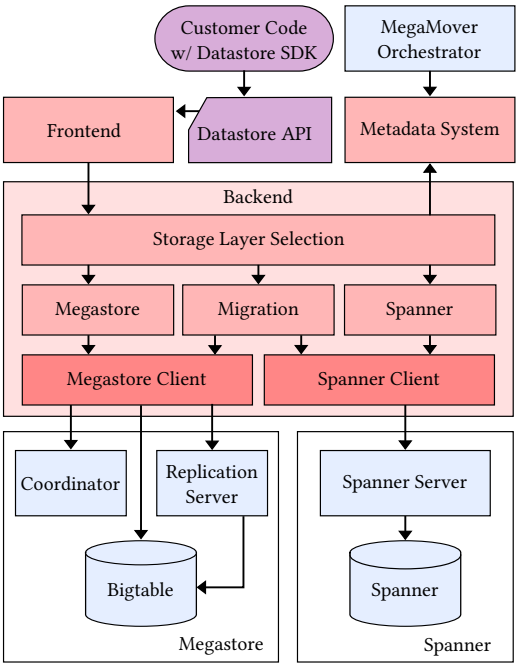


Figure 1: Migration-specific components in the Firestore architecture.

3 ARCHITECTURE

In this section, we present the overall architecture of Firestore during migration.

3.1 Migration-Related Components

An earlier paper [27] presented the overall Firestore architecture with Spanner as its storage layer. Firestore subsumed the existing Datastore service, adding support for the new API and a Spanner-backed storage layer. Firestore is a service hosted in many geographical locations—Figure 1 shows the architecture diagram of Firestore within each location. It includes the legacy components for the Megastore-backed storage layer, but omits components that do not relate to migration. The customer's application code runs on some platform, such as Google Compute Engine, and communicates with Firestore via the Datastore API. Each rectangle—Frontend, Metadata System, Backend, Coordinator, Replication Server, Spanner Server—comprises up to several thousand tasks. Because the Firestore architecture is multi-tenant, a given task may receive and process traffic for any customer database.

Datastore API requests get routed to the Frontend tasks in the region where the database is located, and subsequently to the Backend tasks that translate them into requests to the underlying per-region Spanner or Megastore storage layer. The Backend communicates with Megastore or Spanner via its respective linked library, but the architecture of the underlying implementations differ. The Spanner client communicates with one or more Spanner Server tasks that

manage the complexity associated with distributed storage, e.g. coordinating replication via Paxos. Conversely, the Megastore library accesses the underlying Bigtables directly, in addition to communicating with two support services: the Replication Server, which handles per-replica aspects of Megastore's Paxos implementation, and the Coordinator, which is an in-memory per-replica cache of the Paxos replication state.

The MegaMover Orchestrator updates Firestore's metadata to shepherd databases through the various migration states. The migration state of a given database determines how traffic for that database is routed among the Megastore, Spanner, or migration-specific implementations. Before migration, traffic goes via the (original) Megastore-backed implementation. Once migration for a database starts, traffic is handled by a second, migration-specific implementation. Depending on migration state, some reads may go to Megastore while the others for the same database go to Spanner. Similarly, writes may go to one of Megastore or Spanner, or during some states of migration writes may be attempted on Megastore but retried on Spanner. Once a database has completed migration, all its traffic goes to only Spanner. Section 4 and Table 1 provide further detail on routing of reads and writes.

As is typical in a distributed system, each Backend task transitions a given database to its latest migration state when it discovers that change from the Metadata System. In other words, traffic to a specific database gets rerouted gradually as each Backend task detects state transitions.

3.2 Storage Layout, Reads, and Writes

This section presents the data format in each underlying storage layer. Firestore inherited Datastore's regional and multi-regional configurations–a regional configuration includes replicas in multiple zones [22] and a multi-regional configuration includes replicas in multiple zones within those regions. All the Firestore (Datastore) databases in a given region or multi-region are stored in a single Spanner (Megastore) *alloc* (*instance*), with each alloc (instance) determining the set of replicas. Although not exactly identical, in practical terms the location of replicas in each Spanner alloc and Megastore instance is the same. This not only preserves network distance and latency for applications for accessing their data post-migration but also minimizes the network latency for the actual migration of data from Megastore to Spanner.

3.2.1 Storage Layout. Both Megastore (via Bigtable) and Spanner use sharding to split or merge an instance's rows into *tablets* which are then assigned to individual servers, for sharing and balancing load. Each tablet contains consecutive ranges of the instance's keyed rows; consequently, key ordering is important for locality and overall performance. Thus, to preserve Datastore's performance profile after migration, both implementations must keep essentially the same storage layout and key order.

In both systems, an underlying Entities table in the Spanner alloc or Megastore instance stores payloads of entities keyed by (in order): a database identifier and the entity's key. As a result, all entities for a particular Datastore database are contiguous in both Spanner and Megastore, and all entities within a particular entity group, which share a common (kind, identifier) prefix, are also contiguous, and so on.

TABLE EntityGroups (Megastore only)

🔑 db_id	🔑 entity_group	log
homes	(users, 752)	...

TABLE Entities

🔑 db_id	🔑 entity_group	🔑 path	kind	...
homes	(users, 752)	/users/752	users	...
homes	(users, 752)	/users/752/rooms/den	rooms	...
homes	(users, 752)	/users/752/rooms/kitchen	rooms	...

TABLE Indexes

🔑 db_id	🔑 index	🔑 value	key to **TABLE** Entities
homes	"size" on rooms	100	/users/752/rooms/den
homes	"size" on rooms	200	/users/752/rooms/kitchen

Figure 2: Example storage layout.

Also in both systems, an index entry maps to an individual row in an Indexes table in Spanner or in the Bigtables underlying Megastore. This index row is keyed by (in order): a database identifier, an index identifier, indexed values, and the indexed entity's key. Similar to entities, all index entries for a particular Datastore database are contiguous in Spanner and Megastore. Within a particular index, entries are ordered by indexed values, as required for the index table scans which support query execution.

Spanner and Megastore rows support multiversion concurrency control by allowing each row to contain the values of all recent writes at their associated timestamp; a background garbage collection process removes obsolete data older than a selectable threshold (minutes to hours).

In Megastore, each entity group contains an ordered write-ahead log [37], usually stored in a single row. As discussed in Section 6, we added an optional *entity group simulation* feature to our Spanner layout for applications that depend on Megastore's entity group-based contention behavior.

Figure 2 shows the storage layout for the three entities in the example `(users, 752)` entity group from Section 2; we assume they reside in a Datastore database with an ID `"homes"`. Figure 2 also shows the entries for the built-in index on property `"size"` in kind `"rooms"`. A `'SELECT * FROM rooms WHERE size > 90'` query would read these index entry rows to find matching entity keys, which are then used to fetch the corresponding rows from the Entities table. For Megastore, all mutations to entities in the `(users, 752)` entity group are appended to its write-ahead log.

3.2.2 Writes. Both Spanner and Megastore use Paxos to synchronously replicate write-ahead log entries across replicas. For Megastore, there is a Paxos instance per entity-group, and for Spanner, tablets are automatically assembled into *Paxos groups*. In both systems, the Datastore write transactions can span multiple Paxos groups using 2-phase-commit.

Transactions in Spanner are fully ACID and the write-ahead log entries used in the underlying implementation are invisible to Spanner users. In particular, the effects of a given write are guaranteed to be visible when the transaction finishes committing. This is not the case in Megastore: although Megastore transactions complete once the log has been replicated, the application of the log entry, which involves adding, removing or updating the Bigtable rows for

entities and index entries at the transaction's write timestamp, happens asynchronously. This application happens independently on each Bigtable replica and may be triggered in one of several ways: (1) by the client that initiated the write (typically as an asynchronous operation) (2) by a client performing a strongly-consistent read or write on the entity group (3) by a background process that periodically applies any unapplied writes.

3.2.3 Reads. We use the term "read" to mean both a lookup of an entity by its key and a query that returns zero or more entities. Reads on Spanner are by default strongly consistent, but can also be requested from a recent timestamp within the hour-long window preserved by the garbage collector. In the rest of the paper, we use the term *strong read* to mean strongly consistent read.

Strong reads on Megastore are scoped to a single entity group. These are implemented by *catching-up* one replica by applying all of its log entries, then reading the data at the write timestamp of the last log entry. *Eventual* reads can read from any replica at the current time, and as a result may be missing recent (not yet applied) writes or see the effects of partially-applied log entries, etc.

3.2.4 Safe Time. Datastore maintains a Safe Time microservice that returns, for a given Megastore replica and key range, a *safe time* t_{safe} value, which represents a maximum (conservative) timestamp at which that key range in that replica is up to date. This service is not responsible for *advancing* safe time, only tracking it; safe time advances as a result of logs being applied, as described earlier.

Knowing a safe time for the key range of the entities of a customer's database (which spans many entity groups, so cannot be queried with strong reads) allows us to select and use a timestamp to copy those entities to Spanner without risk of missing any writes.

4 MEGAMOVER

MegaMover is responsible for creating and maintaining a continuously-updated version of the Megastore-resident data in Spanner and for managing the gradual redirection of traffic to the Spanner version. A *move container* (aka container) is a subset of data within a Megastore instance that migrates as a single unit and is defined by a set of key ranges. A container can include anywhere from one to thousands of customer Datastore databases; Section 7.2.2 discusses how databases are picked and assigned to containers.

Each container progresses independently through a sequence of states (Table 1), starting with all traffic and data in Megastore, and ending with all traffic and data in Spanner. The current state for a container and its Datastore databases is stored in the Metadata System, and used to decide how to handle incoming read and write operations. As it is not possible to atomically transition the whole system between states, we enforce that all actors in the system are in no more than two states, and those states must be adjacent. We carefully design the behavior of each state so that correctness is maintained if an arbitrary number of other actors are in either the prior state or the next, but not both of these at the same time.

There are three actors involved in MegaMover: Backends doing reads and writes, *Log Appliers* applying Megastore log entries, and in some states, Flume pipelines (batch jobs) [7]. Table 1 describes how these actors behave in each state, and when the system can transition to the next state (beyond the overarching requirement that all tasks reach one state before any can transition to the next state). Operations on an entity group g rely on a *catch-up* operation, described below, that ensures Spanner has an up-to-date copy of g.

These states can be grouped into two broad phases: *copy and verification* (states through **verification**) and *traffic redirection* (states through **final_sync**). The correspondingly named sections below provide additional details on the behavior of those states; finally, we briefly discuss a cleanup phase that does not feature in this table. But, we first look at how MegaMover extends Megastore's Log Applier system to copy live writes to Spanner.

4.1 Log Applier and Transfer Replicas

MegaMover reuses Megastore's replication system to propagate writes to Spanner. Each container designates a subset of the Megastore instance's replicas as *transfer replicas*. During the migration, when a transfer replica's Log Applier applies its per-entity group write-ahead log entries to its Bigtable rows, it also writes to Spanner[1]. We use multiple transfer replicas for redundancy at a small CPU cost; this allows the migration to continue even when a single transfer replica is unavailable. The form of this transfer depends on the migration state (as detailed below): early on, the log entry is added to a per-entity group journal (stored in Spanner); once it is safe to do so the log entry is applied directly to Spanner. Although we do not present the exact Megastore and Spanner schemas in our paper, the actual data representation in Spanner is substantially different from in Megastore, so this process must also transform the Megastore log entry into the corresponding Spanner mutation.

The catch-up operation ensures that all committed writes on an entity group have propagated to Spanner. It is only used in states where the journal is empty and implemented by performing a strong read on a transfer replica (ignoring the result), thus causing the Log Applier to replicate those committed writes to Spanner.

4.2 Copy And Verification Phase

The copy and verification phase establishes a continuously-updated copy of a container's data in Spanner. This process has two parts: copying the data that exists as of the start of the migration, and applying the writes that occur after migration starts to Spanner.

After initial creation of the container, a *copy timestamp* (t_{copy}) and a *copy replica* are chosen during the **preparing_transfer** state, and added to the container configuration. This timestamp is chosen several minutes in the future to give all actors sufficient time to acknowledge it. If any actor does not acknowledge before that timestamp elapses, this step is repeated with a new copy timestamp; otherwise, the container transitions to the **journal_and_copy** state. Writes committed prior to the copy timestamp are propagated to Spanner via a *bulk copy* Flume pipeline. Before initiating this bulk copy, MegaMover polls the previously described Safe Time service until $t_{\text{safe}} \geq t_{\text{copy}}$ on the copy replica for the container's key ranges. During the **journal_and_copy** state, Megastore garbage collection is configured to preserve the latest version at or before t_{copy} so that it is available for the bulk copy.

[1] Multiple tasks may attempt to apply the same log entry at the same time; these conflicts are resolved by running the apply logic idempotently using a Spanner transaction.

Table 1: MegaMover States and Actor Behaviors

state	next state when	eventual read	strong read of g	write to g	applier of l to g
on_megastore		to Megastore	to Megastore	to Megastore	idle
preparing_transfer	copy timestamp selected				add l to g's journal
journal_and_copy	Flume pipeline copying data as of copy timestamp to Spanner completes				
journal_or_apply	Flume pipeline applying all journaled logs completes				if g's journal is empty: apply l to Spanner otherwise: add l to g's journal
verification	Flume pipeline comparing Spanner and Megastore data succeeds				apply l to Spanner
redirect_eventual	Spanner has adjusted to load	to Spanner			
redirect_strong	Spanner has adjusted to load		catch up g, to Spanner	if g terminated: to Spanner otherwise: to Megastore	
terminate_writes	as soon as possible			catch-up g, terminate g, to Spanner	
final_sync	all journaled logs applied			catch-up g, to Spanner	
on_spanner			to Spanner	to Spanner	idle

Simultaneously, starting in the **preparing_transfer** state, the Log Applier adds all Megastore writes to the per-entity group journal, as applying them to Spanner is not possible while the bulk copy is in progress.

On completion of the bulk copy, the container transitions to the **journal_or_apply** state. In this step, a *bulk apply* Flume pipeline applies the log entries from all the Spanner journals, until all journals are empty. Also from this state on, the Log Applier applies log entries directly to Spanner if the corresponding per entity group journal is empty.

At this point, both Megastore and Spanner must have equivalent copies of the container's data, which we confirm in the **verification** state using a *data verification* Flume pipeline. The job operates in two passes. The first pass does a fast comparison of the container's data in Megastore and Spanner by computing a canonical fingerprint of each entity in each storage system, joining the fingerprints by key, and emitting keys for which one fingerprint is missing or the fingerprints do not match. Because the first pass operates over an inconsistent view (just the most recently applied version of each row) of the data in Megastore, it risks reporting false differences.

The second pass eliminates false positives using the following algorithm on each entity flagged by the first pass. First, perform a strong read of the entity on a transfer replica and note its version. Performing this read on the transfer replica ensures that Spanner is also caught up to this version. Second, read the latest version of the entity from Spanner. Third, perform a second strong read of the entity from Megastore and again note its version. If it matches the version from the first read, then the entities from Spanner and Megastore must match; otherwise, repeat the process.

If the comparison of one or more entities fails repeatedly, the failure details are logged, the migration of the corresponding database is paused, and the engineering team root-causes the underlying problem.

4.3 Traffic Redirection Phase

With the Spanner copy now established and verified, the container starts redirecting traffic. In **redirect_eventual**, eventually consistent reads are redirected to Spanner. Then, in **redirect_strong**, strong reads are also redirected to Spanner (reads within transactions remain on Megastore). In both states, the reads are redirected gradually using a geometric progression to allow Spanner's load-based tablet splitting sufficient time to adjust to the new traffic. The initial redirection includes less than 500 QPS of read traffic; each subsequent redirection occurs at least 5 minutes later and increases the redirection fraction by approximately 50% [19]. Eventually consistent reads are sent directly to Spanner without considering replica freshness. For strong reads, a catch-up of the corresponding entity group is first performed to ensure Spanner is up to date. Up until this point of the migration, Megastore remains the source of truth for that container, and a migration can be easily and quickly undone by redirecting all reads back to Megastore and deleting the copy in Spanner.

Next, the container transitions to the **terminate_writes** state in which writes and transactional reads go directly to Spanner, bypassing Megastore; this is the "point of no return"[2]. Just prior to this transition, the data verification Flume pipeline runs again, using a different Megastore replica than in the **verification** state for its comparison to further harden our data integrity guarantee.

In **terminate_writes**, tasks must first explicitly *terminate* entity groups in Megastore, and tasks that are still in the **redirect_strong** state must detect terminated entity groups and also send their writes and transactions to Spanner. Once all tasks have reached the **terminate_writes** state, which typically happens within approximately 5 minutes, the container transitions to the **final_sync** state, in which the termination of entity groups is no longer required. There may still be unapplied log entries in Megastore which are copied to Spanner during **final_sync**. Only once Megastore's safe

[2]In reality, it's the "point of no easy return"; Section 7.4 discusses this in more detail.

time for the container has advanced past the timestamp of entry to **final_sync** (implying that no unapplied log entries remain) can we transition to the **on_spanner** state in which the migration is essentially complete.

Entity group termination causes extra complexity in the implementation of multi-entity group Datastore transactions in the **redirect_strong** state: they may start a transaction on Megastore on a non-terminated entity group, then find that another entity group in the transaction is terminated. In this case, the transaction must be aborted so the client can retry it. In practice, transactions aborting due to conflicting redirects are limited to the short **terminate_writes** window and have not caused noticeable disruptions to any workloads.

4.4 Cleanup or Undoing Migration

All direct writes to Spanner (starting from when the container switches to **terminate_writes**) are applied asynchronously back to Megastore, for a period of several weeks. This is done specifically to help during undoing migration; more on this in Section 7.4. Once this period ends, the container's data is fully deleted from Megastore.

5 PERFORMANCE: MEGASTORE VS SPANNER

Applications accessing the database must not experience unacceptable performance regression during or after the migration; this section presents how we handled this requirement. We look at read (lookup and query) and write operations, and their interaction with indexes. We defer discussion of transactions to Section 6.

Based on our understanding and analysis of Megastore and Spanner, many potential performance concerns were considered early in the project, and confirmed or rejected using synthetic tests. First, we used microbenchmarks to characterize the relative performance of the two storage layers; the inputs to these tests were based on known concerns from the engineering team. Second, we built a mirroring infrastructure described in Section 7.3.2 to run exploratory performance evaluations on several medium-sized customer and test databases. The traffic to those databases was mirrored to Spanner to provide a more accurate view of potential performance differences. Third, when analysis of logs or other telemetry suggested a specific database would likely experience performance degradation based on a particular traffic pattern (e.g. write rate to a particular kind exceeding a threshold established by a synthetic test), we used this same mirroring infrastructure to verify the suspicion.

Additionally, for databases above a certain size and traffic, we migrated a customer's pre-production databases (identified through a combination of heuristics and direct outreach) before their production databases. Each such migration was also allowed to progress only slowly. This provided the customer sufficient opportunity to study their application performance and inform us if they saw anything of concern. During these "soaks" we inspected our monitoring dashboards for egregious performance problems. Given the wide surface area of the Datastore API and the wide range of applications accessing them, it was not always possible to accurately tell when a performance regression (seen on the dashboard) was of import to the application; customer outreach was sometimes necessary.

5.1 Indexes

Two traffic patterns were identified during early investigation as candidates for causing performance issues with indexes: writes with *high index fanout* and *index lasering*.

5.1.1 Index Fanout. As described earlier: (1) mutation of an entity requires updating all index entries for the mutated properties in that entity and (2) the synchronous part of the write to Megastore updates only the write-ahead log, with entities and index entries updated asynchronously afterwards. Thus, the user-visible latency of a write to Megastore is largely unaffected by the number of index entries updated. However, in the case of a write to Spanner, all entities and index entries are updated synchronously, and therefore the latency of a write is at least as high as the latency of the slowest index update; depending on the size of and traffic to the database, these index entries may reside in few or many tablets. Therefore, writes that mutate large numbers of indexed properties (aka high index fanout) may experience increased latency post-migration due to increased variance.

Although microbenchmarks hinted at potential slowdown for writes that update more than 10 indexed properties, we found most migrated databases saw improved write latency in the aggregate. We mitigated the risk of unacceptable performance degradation by doing a synchronous, best-effort apply of Megastore writes during migration. This ensured the customer would experience the potential increase in write latency prior to the "point of no return". No customer reported this performance regression, and therefore no indexes had to be disabled or deleted. Furthermore, our monitoring did not show this problem in migrated databases.

5.1.2 Index Lasering. Index lasering is a traffic pattern in which monotonically increasing or decreasing values are written at a high rate to an index. For example, lasering can be caused by a data model where the creation timestamp of each entity is recorded as an indexed property. In that case, updates to that property's index always contain a new, never-seen-before value and those updates form a write "hotspot" in the index key-space range. Because the hotspot moves sequentially through a previously empty index key space, the load cannot be effectively split across multiple servers in a range-sharded system. When the aggregate rate of insertions exceeds the capacity of the single server hosting the key range, insertions slow down or even fail. This behavior is discouraged by our best practices [17].

Index lasering is more problematic with Spanner because all index entry updates are committed synchronously. In Megastore, index entries are updated asynchronously, so slowness or errors when the index entries are updated mostly manifest to the user as additional staleness in eventually consistent queries over the lasered index.

Although lasering was investigated early in the project, our initial microbenchmarks did not stress the system enough to reproduce the issue. It wasn't until a customer database experienced performance regressions due to lasering that we recognized its potential impact.

First, we computed the per-kind write rate (a proxy for per-index write rate) by parsing logs, and used this conservative heuristic to identify databases with potential index lasering; the migrations of

these databases were deferred. Next, we used two approaches to reduce the size of this set of databases: we improved the log-parsing tool to compute the actual index entries written, and we extended Firestore's Key Visualizer [21] to provider coverage of index writes in addition to entity writes. Finally, we used three mitigations to unblock the migration of databases with confirmed lasering: (1) we increased the resource—CPU and RAM—allocations for Spanner servers to better handle hotspots, (2) we added a feature to the Datastore API that lets customers disable specific single-property indexes (built and maintained by default) that caused lasering and weren't being used, (3) we built a transparent index sharding system (Section 8.2) to reduce the intensity of lasering.

5.2 Eventually Consistent Reads

By default, Datastore provided strong reads for lookups and a subset of query types and eventually consistent reads otherwise. To improve response latency, users could also explicitly opt into eventual consistency reads for lookups and all queries. Firestore initially provided strong reads for *all* lookups and queries, even for the explicitly eventually consistent ones. This represented a potential latency regression for migrated databases.

Datastore provided no freshness guarantees for eventually consistent reads, but the results were typically stale by at most a few seconds. Spanner supports two methods of eventually consistent reads: *exact staleness* and *bounded staleness*. Although the former is the most performant, it could result in a noticeable change in application behavior post-migration: instead of nearly-up-to-date results in the typical case with occasional higher staleness, reads would return *consistently* stale results. In particular, an eventually consistent read immediately after a write would never yield the latest data on Spanner. For lookups that explicitly opt into eventual consistency, we therefore used bounded staleness with a small upper bound; thus, reads post-migration return data from a recent, but not necessarily the latest, timestamp. Spanner does not currently support bounded staleness for queries, so all queries remain strong. The relative impact to latency is lower due to the (typically) higher overall latency of queries.

6 CONCURRENCY

The Datastore API offers transactions with strict serializability and external consistency [6] built on abstractions offered by the underlying storage system, Megastore or Spanner. The semantics of transactions in both implementations are identical, but non-functional differences between Megastore and Spanner, especially in the area of contention, can lead to problems for applications. This section describes how we preserve contention behavior during migration. We also offer the option of changing contention behavior post-migration for potentially improved performance.

6.1 Transactions: Megastore vs Spanner

Megastore and Spanner's concurrency control differs both in contention granularity, and in the overall contention control mechanism. Megastore's contention control is optimistic [5]; as a result, Datastore users could run long-running (usually) read-only transactions without fear of blocking other read or write operations. Conversely, Spanner's default (and initially only) contention control is pessimistic (using reader-writer locks). Thus a long-running Firestore transaction would prevent concurrent writes. This concern was identified very early in the project and motivated the development of optimistic concurrency controls for both Spanner and Firestore.

Megastore tracks contention at the granularity of the entity group. Two concurrent write transactions contend if there is at least one entity group that they both access. For instance, consider the case where transaction A reads entity `/users/752/rooms/den`, then transaction B commits changes to entity `/users/752/rooms/kitchen`, after which transaction A tries to commit changes to the entity it had read. A's commit will fail even though there's no actual conflict because both transactions operate on the same entity group `(users, 752)`. Spanner's contention control is fine-grained, either with reader-writer locks (when using pessimistic concurrency control) or by tracking accessed key ranges and validating that these key ranges have not changed from their time of access to the chosen commit timestamp (when using optimistic concurrency control).

Some applications depend on the specific behaviors that follow from Megastore's approach to concurrency control. One example is a customer who backs up data using a long-running (say, minutes long) transaction that reads a large entity group in its entirety, stores it on some external system, and then ends with a single write and commit to detect if the entity group has changed during the transaction (in which case the backup of this entity group is retried). Such an implementation depends on the backup transaction not impeding the main user-facing application, and on conflicts being (very likely) resolved in favor of the user-facing application.

6.2 Transactions And Migration

One key benefit of migration to customers is the option to avoid write bottlenecks associated with entity group atomicity. For this reason, Firestore allows customers to choose one of three concurrency modes that best fits their use case: pessimistic concurrency, optimistic concurrency, and a mode (described below) where the entity group behavior is simulated to guarantee backwards compatibility.

Since transactions are optimistic for Datastore on Megastore, a migrated database defaults to optimistic concurrency mode if analysis of request logs has shown that the workloads on the database would not experience differences in contention behavior. Specifically, the analysis must show that no transaction (prior to migration) failed due to overlap with another transaction contending on the same entity group but not the same entities. Otherwise, the migrated database defaults to the entity group simulation mode.

6.2.1 Entity Group Simulation Concurrency Mode. The entity group simulation concurrency mode uses a new Spanner EntityGroups table with one row for each entity group that maintains a log_position column. An entity group simulation transaction uses multiple underlying optimistic Spanner transactions—one per participating entity group—that include the corresponding EntityGroups row. The "log position" of the Megastore write-ahead log is simulated by storing a monotonically increasing number in the log_position column. Thus, Spanner validates at commit time that this simulated log position is unchanged from the time the entity group

was last read by the Datastore transaction, thereby guaranteeing atomicity. Randomized testing was used to ensure Datastore on Megastore concurrency and entity group simulation concurrency behave identically.

No analysis can perfectly tell if a transaction that commits with optimistic concurrency on Spanner but aborts on Megastore changes a behavior the customer application has been relying on. Therefore, customers are given control over their concurrency mode. They can, for example, switch away from entity group simulation to reduce contention, or switch to the conservative entity group simulation mode.

7 TOPICS IN PRACTICE AND INSIGHTS

The scale and scope of this multi-year project required tackling significant practical and operational problems. In this section, we describe the important ones and draw insights that may be useful to other practitioners.

7.1 Insights

Google has an established history of large scale live infrastructure changes [29]. The effort described in this paper benefited from this institutional experience, but several important differences set this migration apart: (1) The migration was automatic, requiring no action from the customer. This gave us substantial control over the project timeline but meant we could not ask customers to make changes to accommodate the migration; we had to identify and support all behaviors of the old system on which customers depended. (2) There were few opportunities for interactive communication with customers. Most customers received exactly one email notification from us, and very few contacted us. It was sometimes difficult to determine if any customers relied on a particular behavior of the old system; when in doubt, we were forced to assume at least one did. The remainder of this section explores the key insights from the Firestore migration planning and execution.

No amount of design, forethought, and testing can guarantee against latent unknowns, and therefore a well-tested capability to undo migration is a must-have fallback option. A lightweight undo capability can enable informed risk taking. We built two mechanisms for undoing migrations: reversion (see Section 7.4.1) and rollback (see Section 7.4.2). When reversion is applied prior to read redirection states, the customer may be fully unaware that part of the migration took place. Having this lightweight, automated capability enabled us to defer development of some pre-migration screening checks (see Section 7.2.3). When a failure occurred during the verification state, we simply reverted that database's migration. Conversely, rollback was treated as a process of last resort.

Analysis of pre- versus post-migration performance is needed both in the aggregate and for specific customer workloads, necessitating testing with standard benchmarks as well as mirroring production traffic. The performance characteristics of a new storage system may be well understood prior to the start of a migration, however synthetic benchmarks alone are insufficient for predicting the performance of real customer workloads. Early in the development process we built infrastructure to duplicate live production traffic (see Section 7.3.2). The practice of workload duplication for evaluating performance has also proven invaluable for other large scale migrations [3].

Explicitly and exhaustively testing state transitions is a useful tool for catching unforeseen corner cases. Large scale live migrations involving distributed systems may involve a large combinatorial number of states and actors. Nevertheless it is critical to build comprehensive test infrastructure to catch issues in early development and to improve developer productivity through easily debuggable and reproducible tests. For the Firestore migration, version skew testing (see Section 7.3) was indispensable for validating that the distributed system behaved correctly as the configuration state evolved. Comprehensive testing over data lifecycles relative to discrete migration states caught subtle correctness issues and race conditions (see Section 7.3.1).

Starting with "low-risk" migrations is a good strategy, but should be balanced with the need for early discovery of unknown problem scenarios. Performing migrations in strict order of "complexity" may reduce the overall risk of failed migrations (see Section 7.2.2), but it misses opportunities to get early insights into upcoming challenges. Identifying customers that are willing to accept slightly higher risk in exchange for early access to the new system enables the development team to apply lessons learned well in advance of the other migrations of similar complexity. Apart from improving the experience for those subsequent customers, this strategy can shorten the overall project timeline.

Large scale migration cannot be achieved without well-oiled automation and tooling to divide the work into configurable batches and orchestrate their progression. To avoid unsustainable toil, any large scale migration requires early investments in automation, observability, and debuggability. To migrate the 1M+ databases we built automation to classify, order, and group databases into move containers (see Section 7.2.1). The team also set up a dashboard to track the in-flight migrations across all locations. Every migration state change was timestamped and tracked in an internal per-database metadata system; this greatly simplified debugging when the timing of events was important for root-causing a problem. The Flume pipelines generated detailed logs and activity reports. The team also wrote and maintained comprehensive playbooks for diagnosing and dealing with known issues.

Even in a transparent migration, customers want observability into the process. Customers recognize that there is risk involved in a migration and want visibility into the process; providing detail builds trust. Timing of state changes is especially helpful as it enables customers to rule out migration as a trigger for unrelated performance changes in their application. Each customer received an automated email before migration with a link to documentation detailing the stages of migration [15] and a list of support channels. The web-based administrative console displayed a banner while migration was in progress. Customers were also able to receive logging-based updates on the state of their migration; these could be used to generate customized notifications to a destination of their choice. For some of the largest customers, a bidirectional line of communication was opened via email; in all but this last case, we could not assume that any of the notifications were actually received or read.

Even in an automatic migration, customers want control of the process. Customers appreciate having some level of control

over the migration process, especially around timing. We provided several such limited channels. Customers above a certain size were able to specify time periods during which migration should not occur; they were able to use this to prevent migration during high-risk periods such as calendar days with expected high traffic or other production changes. Larger customers were also provided with the ability to pause and resume their migration. This did not enable them to revert a migration, but they could prevent it from progressing to the next state. This was primarily to give them time to rule migration out as a trigger for newly-observed (but unrelated) performance issues. This feature did not see wide usage, but several customers expressed appreciation at the existence of the feature. Finally, for a few very large customers, we undertook "high touch" migrations, in which we had a bidirectional line of communication with the customer (usually email; occasionally video chat). This provided the customer with even more precise control over timing and deeper insight into the process. Even in this situation, we didn't require customer-driven application changes.

For long-running projects, setting intermediate milestones and demonstrating incremental progress are important both for internal stakeholders and for team morale. We built internal dashboards that showed time series graphs of the number of databases, total storage size, and QPS on both the old and new storage systems. These numbers were reviewed during internal weekly syncs along with the number of move containers currently in flight. Monthly reports to stakeholders highlighted these numbers and noted geographical locations in which all databases had been migrated. The team also celebrated many informal milestones, such as when QPS on the new system exceeded QPS on the old system. In addition to assuring leadership about the project timeline, these generated a feeling of momentum and accomplishment for the migration team.

7.2 Deconstructing the Problem

Starting in early 2020, all newly-created Datastore databases were backed by Spanner. This prevented the creation of any new databases on Megastore, but not the growth of user traffic or storage consumption of the existing databases. We had to migrate over a million databases that spanned a very large spectrum of QPS, storage size, and customer profiles. Significant automation and tooling was necessary to minimize manual intervention.

7.2.1 Orchestration and Automation. We built an orchestration mechanism to manage the migration workflow of each move container; it allowed us to have multiple containers proceed independently and concurrently through the workflow, and in multiple regions. Additionally, we built automation to order and organize databases into move containers, and then to dispatch each move container into the migration workflow. Although the details of these mechanisms are out of scope for this paper, we discuss here the larger problem of how databases are picked for migration. The choice is driven by multiple factors: the need to keep making steady progress on migration, using early migrations to gain insights into problem scenarios and customer needs, efficiently communicating these changes to customers, but above all minimizing risk to customers.

7.2.2 Database Ordering and Project Milestones. Datastore databases exhibit a large spectrum of sizes, traffic patterns, and feature use. Developers often create small ad-hoc databases for testing, and leave them idle indefinitely. Such "cold" databases are the simplest and least risky to migrate. Conversely, "hot" databases use many features, receive high QPS, and are sensitive to performance regressions, semantic changes, and increased error rates. Enterprise customers often operate multiple databases for different unrelated applications. We also observed that for any given application, it is common to maintain one database for development, one for testing, and one for production. In some cases the naming of the databases betrays this relationship, but conventions vary across our customers. Although it is desirable to migrate the development and test databases before the corresponding production databases to catch potential issues early, it is not practical to solicit input from all customers for the optimal ordering across their databases.

Our overall migration project was divided into milestones sorted by the complexity of the migration. A periodic ordering workflow parsed request logs to generate per-database information such as bytes at rest, QPS, feature usage, concurrent access patterns, billing, and other relevant statistics. The workflow mapped each database to a specific named milestone, such as "Billable databases with less than 10k QPS", or "Databases requiring entity group serializability with less than 500k QPS". Based on the progress of our project, the workflow was configured to admit only databases from milestones that had been already achieved with respect to development work, tooling capabilities, and rigorous testing. If databases from the same customer become eligible for migration together, the workflow also included naming heuristics; all things being equal, a database with "test" or "dev" in the name would need to complete migration before a similarly named database (with or without "prod" in its name). The number of concurrent containers per milestone was also a configurable parameter.

Milestones were determined well in advance of migrations to allow incremental progress. The scope of each milestone was defined in terms of metrics such as peak database QPS, storage size, and workload patterns. The first milestone was limited to "cold" (zero activity in the last 28 days) databases. However, the machinery was also developed and tested to handle a cold database suddenly receiving traffic during its move. We then advanced to "minimal scale" databases with a conservative QPS of < 0.1, minimizing the possibility of contending transactions and allowing us to defer the entity group simulation work (Section 6.2.1). Subsequent milestones increased admissible throughput to 1k QPS, then 10k QPS, and beyond. The experience with each milestone informed the development of subsequent milestones. This methodology ensured forward progress while allowing for incremental learning from failed migrations.

7.2.3 Pre-Migration Screening. Automation sends communication to each customer approximately one week in advance of the migration of any database, which gives them sufficient time to plan and/or respond to us. Customers are sorted based on various criteria, and the largest customers are given the choice to defer their migrations and/or override the proposed ordering for their databases.

A container undergoes additional screening before entering the migration workflow. Each entity in each database is validated by converting it to the Spanner schema format, then back to the Megastore format, and then comparing it with the original entity; this validation is also required to guarantee safe rollback (Section 7.4.2), if needed. When the checks yielded failures for corner-case data formats, they were handled in a subsequent milestone. If an entity in a database fails a check, the database is removed from the container and marked for deferral. In early migrations, such validation did not occur until the **verification** state; any failure would pause the corresponding database's migration and necessitate its removal from the container. Subsequently, we replicated this validation as additional pre-migration screening, which allowed us to learn about corner-case scenarios earlier and to inform customers about migrations that needed to be delayed.

7.3 Testing and Validation

As expected, a strong emphasis on comprehensive testing has been key to success of the project. We developed end-to-end tests to perform migrations in a hermetic test environment. The components running in this test environment were instrumented to induce a migration state skew—tasks of any given component move from one migration state to the next at different times—mimicking the real-world behavior of a distributed Firestore system. Tests single-stepped through the migration state machine, performing database operations and validation before, during, and after each state.

7.3.1 Migration States, Entities, and Transactions. The load generators ensured that in any given migration state, an entity could be either created, updated, deleted, or not accessed. For example, there would be at least one entity created in the **on_megastore** state, and then updated during the **journal_and_copy** state. At least one entity would be created in **preparing_transfer**, and deleted in **journal_and_apply**, and so on. The tests exercised all possible lifecycle permutations. At various points in time, the entity is validated against the expectations for its specific lifecycle and the current migration state. If a particular entity fails the validation checks the test automation freezes the entire testing environment, and preserves the on-disk state for analysis. The failing entity's key captures its specific lifecycle, which greatly aids debugging. The reversion and rollback scenarios (Section 7.4) were similarly tested.

We tested Datastore transactions in a similarly comprehensive fashion. For every adjacent pair of migration states, there is at least one test that initiates a Datastore transaction in the prior state, and attempts to commit the transaction in the subsequent state. Note that the lifetime of a Datastore transaction has an upper limit [23], and that prevents a transaction from spanning more than two adjacent migration states, thereby reducing the combinatorial number of tests. The test matrix covered all possible meaningful sequences of entity reads and writes occurring within a transaction.

7.3.2 Mirroring. Synthetic workloads cannot simulate every combination of database operations, features, loads, and state of the underlying storage system. Therefore, although crucial in evolving our understanding of Spanner performance, and confirming or rejecting many of our designs, they are insufficient for understanding the performance characteristics of real customer workloads.

To address this gap, we developed infrastructure to selectively forward live production traffic to a second set of "mirror" servers. Much like the *tee* command in Unix-like systems, it forwards the requests coming into a Firestore Backend on to a set of secondary Mirror Backends. However, unlike *tee*, requests are duplicated on only a best-effort basis; failures or delays in the mirrored requests have zero impact on the customer production traffic.

The mirrored traffic enters a hermetic and ephemeral mirror environment that is fully isolated from other production systems except for a shared request log collection service. An on-demand workflow stands up an instance of this environment, and then enables traffic mirroring for the chosen set of databases. For a given experiment, several hours of warm-up time is required to populate a sufficiently large subset of the database. This is followed by several days of performance monitoring via the request logs. To get a high quality analysis, we require a large representative sample of request types, and that a sufficiently large percentage of those requests were processed on the mirror. At the completion of an experiment, the environment is deleted, along with all the mirrored data.

The original and mirror logs contain unique identifiers that allow individual requests to be matched one-to-one, allowing an apple-to-apples comparison of operation latencies, error rates, and other interesting metrics. The lack of true parity in the data sets and that a small fraction of requests are not forwarded do not significantly impact the quality of the resulting analyses. We made extensive use of this infrastructure to gain insights into customer traffic well before their scheduled migrations. The resultant analyses helped us study, test and tune several aspects of performance, such as index lasering, the performance of entity group simulation on Spanner, and automatic sharded indexes. From these results, we identified potential performance issues in databases belonging to important customers so their migrations could be deferred until the issues were mitigated or fixed.

7.4 Undoing Migration

We distinguish between a *reversion*, which is a zero-downtime backward MegaMover state transition, and a *rollback*, which is a full reverse migration after the normal migration to Spanner has completed. We designed reversion such that the migration of a subset of databases in a move container can be undone even while the rest of the container continues its migration. Conversely, rollback executes on an individual database.

7.4.1 Reversion. Reversions are trivial in the copy and verification phase—the migration can be terminated by stopping the transfer and journaling of writes to Spanner. Since Megastore remains the source of truth, there is no user-visible impact from aborting the migration at this stage. In practice, such reversions took place if the verification job identified data mismatches between Megastore and Spanner after the copy was complete and the journal was drained.

Reversion is also possible in the read traffic redirection states (**redirect_eventual** and **redirect_strong**) by following the above procedure after first transitioning back to the **journal_or_apply** state. In practice, a reversion at this point took place if the system detected a performance regression in read latency on Spanner.

7.4.2 Rollback. After write traffic is redirected to Spanner, reversion is no longer possible since the data in the two storage systems are no longer identical. To deal with this contingency, we engineered rollback that undoes migration after this "point of no return". Crucially, rollback provides a safety net to handle latent unknowns post migration.

Rollback may be needed from two distinct starting points: before or after Cleanup. Because rollback is easier in the former—the source data in Megastore is still available—we configure a multi-week delay before Cleanup (see Section 4.4) to make the latter unlikely.

Before Cleanup: When MegaMover begins directing write traffic to Spanner, we maintain a record of the Spanner-only writes in a per-database Spanner queue. Rather than recording the entire payload of each write, this TransferToMegastore queue records (in the same transaction as the write) the key of any entity that has been modified, inserted, or deleted, and the commit timestamp of the last update. An asynchronous processor receives updates from this queue, reads the latest state for the relevant entities from Spanner, and copies it back to Megastore. It batches updates by entity group to optimize the Megastore writes. This process continues during the multi-week period before Cleanup occurs. The following steps execute a database rollback. First, disable write operations (reads are still allowed). Second, drain the TransferToMegastore queue—no writes means no entries are being added to the queue. Third, redirect read traffic to Megastore. Fourth, enable writes once the configuration state indicating Megastore as the new source of truth has propagated to all actors. We found experimentally that the queue remains nearly empty in steady state—the processor has to transfer only the latest state of each entity back to Megastore, making it possible to ignore intermediate updates within the batch.

After Cleanup: We also built a slower rollback for the unlikely event in which a database has to return to Megastore after data there has already been deleted. This procedure repeatedly copies all data back with differential snapshots, and disables writes for the last copy when the differences are sufficiently small. The write downtime for this procedure is proportional to the update rate of the migrating database; it was never required outside of testing.

8 EVALUATION

We first present some production data to convey the scale of this project, and then discuss performance of sharded indexes.

8.1 Production Data

All (over one million) databases have been successfully migrated in 20 locations. The migrated databases covered a very large range of storage size and QPS: among databases that exceeded 1 QPS, the largest by storage was 800,000 times larger than the median; the most active had 70,000 times more QPS than the median. At peak, over 71,000 databases were migrating concurrently.

8.1.1 Concurrency Modes. As described earlier, automation selects the appropriate concurrency mode based on usage statistics and write patterns. Until optimistic concurrency was available in Spanner, we migrated only databases with less than 0.1 QPS of traffic with pessimistic concurrency mode, which is the default for Firestore databases. The low traffic reduces the likelihood the lock contention that is more likely in this mode. This constituted 29.4% of all databases migrated.

Once optimistic concurrency was available, all databases were allowed to migrate, using either optimistic concurrency control (69.8%) or entity group simulation (0.8%). These modes better match the behavior of Megastore transactions: writers do not block readers, conflicting writes fail on commit rather than being delayed on lock acquisition. As explained earlier, traffic analysis picked entity group simulation for the databases that showed concurrent transactions accessing different entities in the same entity group. For only 0.2% of these databases did customers subsequently choose to switch to either optimistic or pessimistic concurrency.

8.1.2 Undoing Migration. As described earlier, reversion is simpler than rollback and requires no downtime. Approximately 16K databases in 85 containers experienced at least one reversion. The primary reasons for reversion were: failed data integrity checks, read performance problems, and write performance problems.

Whenever the verification job fails, the migration is undone, the migrated data purged, and migration is restarted after the underlying issue has been debugged and fixed. As expected, once we replicated the data validation step as a pre-migration screening step (Section 7.2.3), such cases were caught earlier and the number of reversions due to failed data validation dropped to zero. Reversions for the other two reasons were triggered if a read performance regression was seen during the read redirection states or write performance regression on the asynchronous write path during and after the **journal_and_copy** state.

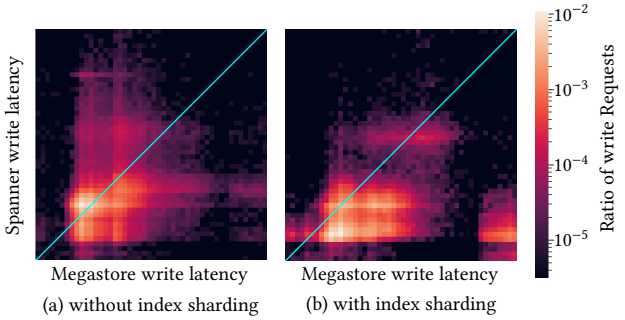
Rollback was required for only two databases that belonged to the same customer, and fortunately before Cleanup. A customer experienced a novel performance regression that surfaced weeks after the migration. First, we performed a rollback of the customer's staging database, which had already been migrated. We then repeated the procedure on the customer's production database, timed to coincide with a weekly low point in the customer's traffic. The customer was involved in this planning. The database experienced no read outage and only 15 minutes of write downtime.

8.2 Index Sharding Performance

As described in Section 5.1.2, writes with monotonically increasing or decreasing values may cause index lasering, which is more problematic with Spanner because index entry updates are applied synchronously. Based on an analysis of request logs, the migrations of 134 customer databases were deferred. After the steps mentioned in Section 5.1.2, fewer than 30 databases needed index sharding.

8.2.1 Mirroring Sharded Indexes. We used mirroring to compare Megastore with Spanner for customer workloads with index lasering. Figure 3(a) compares the Megastore and Spanner latency for a customer database with several lasered indexes, with peaks of >5k writes/sec. The bright white area just above the diagonal blue line shows Spanner's comparative slowness for typical lasering writes. The dim magenta area high above the diagonal line shows Spanner's long performance tail under intense lasering.

Transparent index sharding partitions a single laser into N lasers of $1/N$ intensity each, potentially served by different Spanner servers. We inserted a shard identifier (with N possible values) into



(a) without index sharding (b) with index sharding

Figure 3: Write latency with index lasering. X and Y axes are log scale in milliseconds. Each pixel represents a fraction of customer writes comparing their latency on Megastore vs the Spanner mirror. The pixel color indicates the percentage of the total writes plotted in the graph.

the index row key before the index identifier and the (monotonic) indexed values, producing an index row key with the following elements in order: a database identifier, a shard identifier, an index identifier, the indexed values, and the indexed entity's key.

Figure 3(b) shows the result from mirroring a second customer database that lasered several indexes, with higher (>10k peak) writes/sec. The second database is otherwise very similar to the first; in fact they are a customer's production and pre-production database, respectively. Each index was sharded 10-way in the mirror. Note that despite the more intense lasering most of the brightly-colored area above the diagonal from Figure 3(a) is now below the diagonal (Spanner is faster) in Figure 3(b).

8.2.2 Tuning Index Sharding. We chose the shard count based on the customer's peak index write rate. Although sharding mitigates lasering it also slows down queries, which must now read all shards when searching an index. Based on internal experimentation, 500 writes per index shard minimizes impact on query latency while providing headroom for future increases in index lasering intensity. After picking an initial shard count for a customer database, we further tuned it based on analysis via mirroring.

To better understand query performance regression, we examined customer queries with a LIMIT clause of 1 through 10; note that these lead to the worst-case ratio of rows in the result set to number of distinct shard reads. Compared to other databases in the same region that did not require sharding of their indexes, queries to those with sharded indexes were 41% (p50) and 40% (p99) slower—an acceptable slowdown for these customers.

An early prototype derived the shard identifier from the unsharded index row key. However, we discovered that databases with lasering behavior often laser on multiple indexes (on different properties). Thus a write of a single entity would often update different shards (on different Spanner servers) for each index that needed to be kept up to date. We mitigated this problem by deriving the shard identifier from the entity row key and by using the same shard count for all single-property indexes in a database. Thus, updates to all indexes for a single entity are closer in index key-space and handled by a smaller set of Spanner servers that owns the index shard key-ranges for that entity. This change improved p99.9 write latency by almost 50% when compared to the earlier prototype.

9 RELATED WORK

There has been an industry-wide trend in the migration of workloads from data centers to the cloud, and this includes databases as well. Major cloud providers such as AWS [1], GCP [14], Azure [31], and OCI [33] have all published white papers on methodology and tooling for both bulk and streaming data migrations into their cloud services. For databases, the focus has been on offline ETL-style data moves and data transforms for migrating from an RDMBS to Document databases [26, 35], or migrating between different flavors of NoSQL databases [36].

Tools such as Mongify [9] and Apache Sqoop [13] have been developed to facilitate large bulk migrations from RDBMS to NoSQL databases, using a snapshot of the source data. For moves to the cloud, database migration services for GCP [20] and AWS [2] offer lift and shift solutions with low downtime. For live migrations, there has been research on live data replication using differential snapshots [12], log replication using efficient stream processing frameworks [30] and stream processing combined with relational to document format transforms [32]. The migration machinery presented in this paper uses an approach similar to a bulk snapshot copy combined with active log replication, however it provides a critical orchestration layer, which manages the customer's write and read traffic during the migration.

For emergency rollback after migration, the fast rollback method described previously (Section 7.4.1) uses an approach similar to trigger-based replication [25], while the slower rollback method is analogous to differential snapshots [12].

The survey of existing migration tooling and literature reveals a widespread business need for large scale database migrations, including moves from RDBMS to NoSQL databases. Research on data replication has generated techniques for fast bulk migrations with efficient live synchronization of database state. The machinery presented in this paper uses a replication model similar to documented techniques, while leveraging unique architectural features of the Megastore storage system. Unlike many of the migration solutions described in literature, the application data model is preserved across the migration, along with the transactional semantics of the source database. Migrating customers enjoy a zero-downtime, fully managed migration experience that requires no intervention.

10 CONCLUSION

In this paper, we presented the multi-year process of non-disruptive migration of more than a million Datastore databases from Megastore to Firestore's Spanner-backed storage system; all applications retained access via the Datastore API with zero downtime during and after migration. We also presented the changes we made to Firestore, in particular in the domain of concurrency control, to ensure that all customers migrated to an implementation that was not just better overall, but better for their particular workload.

ACKNOWLEDGMENTS

We thank the many Googlers and ex-Googlers who worked to make this project a success.

REFERENCES

[1] Amazon. 2016. An Overview of AWS Cloud Data Migration Services. https://docs.aws.amazon.com/whitepapers/latest/overview-aws-cloud-data-migration-services/overview-aws-cloud-data-migration-services.html. Accessed 2024-07-12.

[2] Amazon. 2016. AWS Database Migration Service. https://aws.amazon.com/dms/. Accessed 2024-07-12.

[3] Ankita Girish Wagh. 2022. Online Data Migration from HBase to TiDB with Zero Downtime. https://medium.com/pinterest-engineering/online-data-migration-from-hbase-to-tidb-with-zero-downtime-43f0fb474b84. Accessed 2024-07-12.

[4] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 223–234. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[5] Philip A. Bernstein and Nathan Goodman. 1986. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13 (1986), 185–221. https://api.semanticscholar.org/CorpusID:30874

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA.

[7] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 363–375. http://dl.acm.org/citation.cfm?id=1806638

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26. https://doi.org/10.1145/1365815.1365816

[9] Anlek Consulting. 2011. Mongify. https://github.com/anlek/mongify.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[11] Ed Davisson, Tilo Dickopp, David Gay, Eric Karasuda, Ram Kesavan, and Vadim Yushprakh. 2024. Transparent Migration from Datastore to Firestore (Extended Version). https://research.google.com.

[12] Wei Du and Xianxia Zou. 2015. Differential snapshot algorithms based on Hadoop MapReduce. In *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. 1203–1208. https://doi.org/10.1109/FSKD.2015.7382113

[13] Apache Software Foundation. 2012. Apache Sqoop. https://sqoop.apache.org/.

[14] Google. [n.d.]. A CIO's Guide to Application Migration. https://services.google.com/fh/files/misc/cio_guide_to_application_migraton.pdf. Accessed 2024-07-12.

[15] Google. [n.d.]. Automatic Upgrade to Firestore. https://cloud.google.com/datastore/docs/upgrade-to-firestore. Accessed 2024-07-12.

[16] Google. [n.d.]. Cloud Datastore Transactions. https://cloud.google.com/datastore/docs/concepts/cloud-datastore-transactions. Accessed 2024-07-12.

[17] Google. [n.d.]. Datastore Best Practices. https://cloud.google.com/datastore/docs/best-practices#indexes. Accessed 2024-07-12.

[18] Google. [n.d.]. Firestore API. https://cloud.google.com/firestore/docs/reference/rpc. Accessed 2024-07-12.

[19] Google. [n.d.]. Firestore: Ramping up traffic. https://firebase.google.com/docs/firestore/best-practices#ramping_up_traffic. Accessed 2024-07-12.

[20] Google. [n.d.]. Overview of Database Migration Service. https://cloud.google.com/database-migration/docs/overview. Accessed 2024-07-12.

[21] Google. [n.d.]. Overview of Key Visualizer. https://cloud.google.com/firestore/docs/key-visualizer. Accessed 2024-07-12.

[22] Google. [n.d.]. Regions and Zones. https://cloud.google.com/compute/docs/regions-zones. Accessed 2024-07-12.

[23] Google. [n.d.]. Transactions. https://cloud.google.com/datastore/docs/concepts/transactions. Accessed 2024-07-12.

[24] Google. 2019. NoSQL for the serverless age: Announcing Cloud Firestore general availability and updates. https://cloud.google.com/blog/products/databases/announcing-cloud-firestore-general-availability-and-updates. Accessed 2024-07-12.

[25] Yong Hu and Stefan Dessloch. 2014. Extracting deltas from column oriented NoSQL databases for different incremental applications and diverse data targets. *Data & Knowledge Engineering* 93 (2014), 42–59. https://doi.org/10.1016/j.datak.2014.07.002

[26] Girts Karnitis and Guntis Arnicans. 2015. Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation. In *2015 7th International Conference on Computational Intelligence, Communication Systems and Networks*. 113–118. https://doi.org/10.1109/CICSyN.2015.30

[27] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C. Mohan. 2023. Firestore: The NoSQL Serverless Database for the Application Developer. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3367–3379.

[28] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

[29] W. Look, M. Dallman, and an O'Reilly Media Company Safari. 2019. *Case Studies in Infrastructure Change Management*. O'Reilly Media, Incorporated. https://books.google.com/books?id=uUY7zQEACAAJ

[30] Kun Ma and Bo Yang. 2015. Live Data Replication Approach from Relational Tables to Schema-Free Collections Using Stream Processing Framework. In *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. 26–31. https://doi.org/10.1109/3PGCIC.2015.64

[31] Microsoft. [n.d.]. Cloud Migration Simplified: A Guide for Migrating Infrastructure, Databases, and Applications. https://clouddamcdnprodep.azureedge.net/gdc/gdcOB3Q4w/original. Accessed 2024-07-12.

[32] Basant Namdeo and Ugrasen Suman. 2021. A Model for Relational to NoSQL database Migration: Snapshot-Live Stream Db Migration Model. In *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Vol. 1. 199–204. https://doi.org/10.1109/ICACCS51430.2021.9441829

[33] Oracle. [n.d.]. Oracle Cloud Infrastructure Database Migration (DMS). https://www.oracle.com/a/ocom/docs/oci-database-migration-onprem-adb.pdf. Accessed 2024-07-12.

[34] Charles R. Severance. 2009. *Using Google App Engine - start building and running web apps on Google's infrastructure*. O'Reilly. http://www.oreilly.de/catalog/9780596800697/index.html

[35] Wilson Tandya and Fazat Nur Azizah. 2023. Migration of Relational Database to NoSQL Document-Oriented Database. In *2023 IEEE International Conference on Data and Software Engineering (ICoDSE)*. 180–185. https://doi.org/10.1109/ICoDSE59534.2023.10291584

[36] Yansyah Saputra Wijaya and Arry Akhmad Arman. 2018. A Framework for Data Migration Between Different Datastore of NoSQL Database. In *2018 International Conference on ICT for Smart Society (ICISS)*. 1–6. https://doi.org/10.1109/ICTSS.2018.8549944

[37] Wikipedia contributors. [n.d.]. Write-ahead Logging. https://en.wikipedia.org/wiki/Write-ahead_logging. Accessed 2024-07-12.