2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00259

Firestore: The NoSQL Serverless Database for the Application Developer

Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah
Google
ram.kesavan@gmail.com {dgay, danthev, jimit}@google.com

C. Mohan[*]
Tsinghua University
seemohan@gmail.com

***Abstract*—The recent years have seen an explosive growth in web and mobile application development. Such applications typically have rapid development cycles, and their developers expect mobile-friendly features and serverless characteristics such as rapid deployment capabilities (with minimal initialization), scalability to handle workload spikes, and flexible pay-as-you-go billing. Google's Firestore is a NoSQL serverless database with real-time notification capability, and together with the Firebase ecosystem greatly simplifies common app development challenges by letting application developers focus primarily on their business logic and user experience. This paper presents the Firestore architecture, how it satisfies the aforementioned requirements, and how its real-time notification system works in tandem with Firebase client libraries to allow mobile applications to provide a smooth user experience even in the presence of network connectivity issues.**

***Index Terms*—cloud, database, mobile and web applications, continuous queries**

I. Introduction

A large amount of modern computing happens at the edge, in web browsers or on mobile devices. Deploying applications to such devices is generally easy, either via static hosting or via Google and Apple's application stores. However, many applications need some remote computing and storage, be it just for reliability or state sharing across multiple devices, for sharing information between users, or for querying datasets that are being updated by other processes. Implementing, deploying, scaling and managing this remote infrastructure remains a significant challenge, even in today's world of ubiquitous cloud services.

Firestore is a schemaless serverless database with real-time notification capabilities that greatly simplifies the development of web and mobile applications. It scales to millions of queries per second and petabytes of stored data; notable current and past users include the New York Times and BeReal, as well as a prominent social media app and a mobile game each with over a hundred million users. Importantly, at low scale QPS (queries per second) and storage consumption, Firestore costs close to nothing.

In this paper, we describe four key aspects to Firestore's success and describe how Firestore achieves them.

Ease of Use: Modern application development benefits from rapid iteration and deployment to production. Firestore's schemaless data model, ACID transactions, strong consistency, and index-everything default means that developers can focus more on the data they wish to store and present to the end user without worrying about the details of the database configuration.

Fully Serverless Operation and Rapid Scaling: Some applications go viral, and that translates to difficult problems around scaling of the infrastructure with increasing QPS load, storage, and therefore costs. Firestore is truly serverless–the application developer needs to only create a (static) web page or an application, and initialize a Firestore database to enable end users to store and share data. End-user database requests are routed directly to Firestore, without the need for a dedicated server to perform access control thanks to security rules set by the developer. Firestore's API encourages usage that scales independently of the database size and traffic, and Firestore's implementation leverages Google's infrastructure (in particular, Spanner) to provide a highly available and strongly consistent database whose scale is limited only by the physical constraints of a cloud region's datacenters. Firestore's serverless pay-as-you-go pricing together with a daily free quota ensures that billing increases reflect application success; a standalone emulator allows developers to safely experiment.

Flexible, Efficient Real-time Queries: An application often needs to send fast notifications to potentially large subsets of web or mobile devices for many reasons, such as communication between users. In a Firestore-based application, these are typically coded as real-time (also known as streaming or continuous) queries [1] to the backend database. The results of a real-time query are updated by the application and presented to the end-user to reflect any pertinent change in the database. Firestore supports queries that can be efficiently executed using secondary indexes and updated in real-time from the database's write log (this a key element of Firestore's scalability). These queries fall short of full SQL support, but are generally sufficient for the querying needs of interactive applications.

Disconnected Operation: Mobile devices can lose network connectivity for arbitrary lengths of time. The behavior of an application while the device is disconnected can often be key to its success. The Firestore database service together with Firebase client-side SDK libraries support fully disconnected operation, with automatic reconciliation on reconnection. This greatly simplifies development of mobile applications.

[*]The author was a Visiting Researcher at Google, and is partially supported by Project 62021002 of NSFC.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

This combination has led to a large, thriving community of more than a quarter million monthly-active application developers using Firestore, and to Firestore-based applications directly serving requests to more than one billion end-users on a monthly basis.

This paper makes the following contributions:

- It describes the Firestore data model and simple query language that are key to enabling rapid application development.
- It shows an effective approach for building a multi-tenant database service over a single underlying (scalable) relational database (Spanner). This multi-tenant approach is critical for lowering per-database cost, and thus supporting a generous free tier for millions of databases.
- It outlines how the multi-tenant architecture supports rapid scaling, high availability, data integrity, and isolation.
- It describes how client-side SDK libraries enable disconnected operation, how Firestore computes and fans out notification updates for real-time queries to clients, and how these updates get presented to the end-user in a consistent fashion.

The rest of the paper is organized as follows: Section II provides some history. Section III describes how Firestore gets used—its data model, SDKs, and APIs. Section IV explains how the internal architecture achieves Firestore's various requirements. Section V presents some production data and experimental results (including YCSB benchmarks). Section VI describes some lessons learned in practice, Section VII elaborates on related work, Section VIII discusses future plans and Section IX concludes the paper.

II. History

Firestore grew out of App Engine's [2] Datastore database, launched in 2008. Datastore was accessible only from App Engine and provided a schemaless data model, restricted transactions and flexible queries. It was built on top of Megastore [3], which in turn was built over Bigtable [4], and this architecture forced most queries to be eventually consistent, complicating application development.

Datastore provided a Paxos-replicated option for increased reliability in 2011, and migrated existing users to the new system in 2012. The server-side of a well-known social media app that quickly scaled to many millions of users was an important early customer of the combined App Engine/Datastore system. The Cloud Datastore API was launched in 2013, allowing access to Datastore from outside the App Engine environment. A popular billion-dollar mobile game was launched in 2016 using Cloud Datastore, with traffic rapidly scaling to 10 times the initial "worst case" estimate without significant database scaling challenges.

In 2014, Google acquired Firebase. Firebase's serverless and ease-of-use emphasis matched Datastore's, and that kicked off a project to mesh the best features of Firebase's Realtime Database (RTDB) with those of Datastore. There was also a desire to build this on Spanner [5] to benefit from Spanner's unrestricted transaction capabilities and strong consistency guarantees. This resulted in a new product that launched in 2019 called Firestore: an API that supports real-time queries, fully serverless operation, and disconnected client operation. This is the first paper on Firestore.

The non-disruptive migration of all Datastore databases (on Megastore) over to Firestore's Spanner-backed layout was started in 2021; all applications retain access to their databases via the Datastore API, and with no down time. This migration is expected to complete in 2023. The design and details of that migration are outside the scope of this paper. Both Firestore and Datastore have a common data model, and provide similar access to the underlying data—Firestore calls them documents and Datastore calls them entities—except for the APIs necessary for real-time queries that are exclusive to the Firestore API. Additionally, both APIs can be used to read from and write to the same database. Although all new feature work (that is not related to real-time queries) is added to both APIs, this paper focuses only on the Firestore API to keep the terminology consistent when discussing real-time queries; the term "entity" is used occasionally for historical reasons, and the reader can safely replace each occurrence with "document".

III. Using Firestore

Firestore is a schemaless, serverless, document database offering on Google Cloud that can be initialized as a regional or multi-regional service, where the latter provides higher availability. Section V-D discusses how easy it is for the customer[1] to set up and use the service. This section describes how the application developer interacts with the service.

Throughout this paper we will use examples derived from the Firestore Web Codelab [6]—a functional serverless restaurant recommendation web application, which lets users see a list of restaurants with filtering and sorting, and view and add reviews.

A. Data Model

Firestore supports a rich set of primitive and complex data types, such as maps and arrays. Each *document* is identified by a string, and is essentially a set of key-value pairs that add up to at most 1MiB. Documents can be arranged in hierarchically-nested *collections*. The combination of the collection name and the identifying string forms the document's unique *name* (key). For instance, `/restaurants/one` is the name of a document in collection `/restaurants` with the identifying string `one`. A document can also have a sub-collection, so `/restaurants/one/ratings/2` is a document in collection `/restaurants/one/ratings` with identifying string `2`. Figure 1 shows an example of a restaurant document and Figure 2 a rating document in the sub-collection. Each key-value pair in a document is called a *field*.

[1]In this paper, the term customer can mean one developer deploying their app or a larger enterprise hiring teams of developers that build, deploy, and sustain multiple apps.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

```
/restaurants/one
  address: "415 Main Street",
  type = "BBQ",
  avgRating: 3.5,
  numRatings: 10,
```

Fig. 1: Document example for a restaurant.

```
/restaurants/one/ratings/2
  rating: 3,
  userId: "UUU",
  details: {
    text: "Food was tasty but cold",
    price: "good"
  }
  time: 2022/09/01 13:23:22 GMT
```

Fig. 2: Document example for a restaurant rating.

B. Indexes

To scale with increase in database size, Firestore executes all queries using secondary indexes. To reduce the burden of index management, Firestore automatically defines an ascending and a descending index on each field across all documents on a per-collection basis. The automatically defined indexes for the document in Figure 2 are on `rating`, `userId`, `details`, `details.text`, `details.price`, and `time`.

Automatically defining indexes simplifies development but introduces some risks. First, a write operation becomes more expensive because it needs to update more indexes, which in turn increases latency and storage cost. Second, fields with sequentially increasing values, such as `time` in Figure 2, introduce hotspots that limit maximum write throughput. To address these issues, Firestore allows the customer to specify fields to exclude from automatic indexing (queries that would need the excluded index then fail).

Finally, the customer can define indexes across multiple fields, e.g., rating **asc** and time **desc** to support queries like

```
select * from /restaurants/one/ratings
where rating = 3 order by time desc.
```

C. Querying

Firestore supports point-in-time queries that are either strongly-consistent or from a recent timestamp, and strongly-consistent real-time queries. Both modes support the same query features: projections, predicate comparisons with a constant, conjunctions, orders, limits, offsets. A query can have at most one inequality predicate, which must match the first sort order. These restrictions allow Firestore's queries to be directly satisfied from its secondary indexes.

A real-time query reports a series of timestamped snapshots, where each snapshot is the strongly-consistent result of the query at that specific time. Snapshots are reported as deltas (documents added, removed, and modified) from the previous snapshot. Firestore does not guarantee reporting every snapshot of a query's result, e.g., if the query's results went through the following sequence (@ indicates the document's timestamp):

```
t=10:{/restaurants/one,avgRating:3,...}@1,
 {/restaurants/two,avgRating:4,...}@3
t=13:{/restaurants/one,avgRating:3,...}@1,
 {/restaurants/two,avgRating:3.5,...}@13
t=19:delete /restaurants/one@19,
 {/restaurants/two,avgRating: 3.5,...}@13
```

Firestore might report only snapshots time=10 and time=19, but skip reporting time=13. This flexibility in what snapshots to report gives Firestore more options on how to execute real-time queries but does not impact the typical application which aims to display the latest accurate snapshot of a query's results.

D. Server SDKs

As Figure 4 shows, Firestore has two categories of SDK libraries, each with support for multiple programming languages: "Server" used by applications that run in privileged environments, such as GCE, GKE, Cloud Run, App Engine, and "Mobile and Web" used by applications that run on end-user devices such as mobile phones and browsers. The former category includes older Datastore-API-based Server SDKs.

The Server SDKs map Firestore's data model to the target language, and provide convenient transaction abstractions, such as automatic retry with backoff.

E. Mobile and Web SDKs

One of Firestore's differentiating features is that it allows direct third-party (end-user) access, including support for disconnected operation. This feature is made possible by a combination of: abstractions within the SDKs (in particular, real-time queries), Firebase Authentication [7] which supports end-user authentication from a wide variety of identity providers (Google, Apple, Facebook, phone numbers, anonymous, etc), Firebase Security Rules [8] which is a security language for expressing fine-grained access controls, and the Firestore API.

A developer can structure their application to authenticate users with their choice of identity provider(s), and then program their application using the abstractions provided by the Mobile and Web SDKs. In a typical application, the main abstractions are real-time queries to fetch the state to display and various database updates to reflect the end-user's actions. The direct update of displayed state based on the results of real-time queries greatly simplifies application development: it displays the initial state when the application is opened, it automatically updates the display when some other user changes the state, it also automatically updates the display when this end-user updates the state (avoiding the need for any update-specific display logic), it behaves reasonably when the end-user is disconnected (local updates are seen), and it automatically reflects the results of reconciliation after reconnection.

The SDKs support transactional writes based on optimistic concurrency control while connected, and blind writes at all times. With transactions, all data read by the transaction is revalidated for freshness at the time of the commit; the transaction is retried if the data fails the freshness check. We do not support third-party, lock-based pessimistic concurrency control, because it would allow a third-party to easily conduct

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

```
match /restaurants/{r}/ratings/{s} {
  allow read: if request.auth != null;
  allow create: if request.auth != null &&
    request.auth.uid ==
    request.resource.data.userId;
}
```

Fig. 3: Rating security rules.

a denial-of-service attack on writes to a Firestore database, e.g., by holding read-locks on important documents for a long-time. A blind write's "last update wins" model works well with potentially-disconnected operation, but still requires significant SDK support which is discussed in Section IV-E.

In a system that allows direct third-party access, data needs to be secured at a finer granularity than the whole database to prevent accidental or malicious updates/views. These restrictions are expressed by the customer using Firestore security rules. The example in Figure 3 allows any authenticated end-user to read a restaurant rating, and any authenticated end-user to add a restaurant rating as long as they attach their own user ID to the rating. Updates and deletes of ratings are not allowed.

The grammar allows nesting of match statements and wildcards to simplify writing of rules for sub-collections. The *if* condition can not only check the fields of accessed documents, but also fetch and inspect fields of other database documents (e.g., check an access control list). These additional document lookups are executed in a transactionally-consistent fashion with the operation being authorized.

F. Write Triggers

Firestore allows the definition of triggers on database changes that call specific handlers in Google Cloud Functions [9]. The application developer can define follow-up actions in those handlers based on the changes to the database; the delta from that change is conveniently available in the handler. This supports processing that would otherwise be insecure or too expensive to perform on the end-user device.

IV. Architecture and Implementation

The customer application in Figure 4 may be running on a mobile device, a web client, a VM, or a container [10]. The Firestore service is available in several geographical regions of the world; a customer picks the location of a database at creation time. Each of the four rectangles—Frontend, Backend, Metadata Cache, and Real-time Cache—comprises up to thousands of tasks that get created in each region served by Firestore. Firestore RPCs from the application get routed and distributed across 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 databases.

In this section, we focus on seven distinctive aspects of the architecture that are critical to the serverless nature of Firestore and its ease-of-use: global routing, billing and the free quota, multi-tenancy, writes, queries, real-time queries, and disconnected operation. We skip other aspects, such as

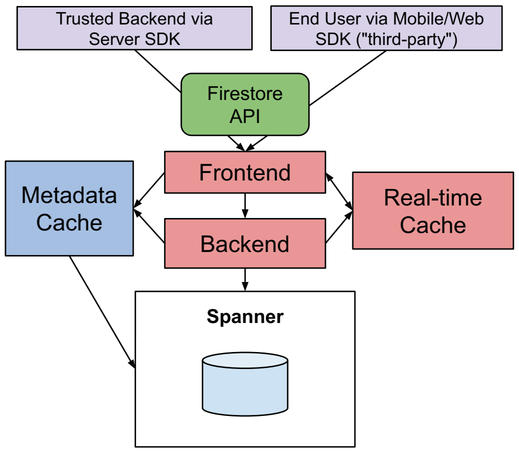

Trusted Backend via Server SDK
End User via Mobile/Web SDK ("third-party")
Firestore API
Frontend
Metadata Cache
Backend
Real-time Cache
Spanner

Fig. 4: Simplified architecture diagram of the Firestore stack.

load-balancing and monitoring, that are not as unique to Firestore compared to other cloud services.

A. Global Routing

Global routing is more important for Firestore than a typical cloud database, as many Firestore requests originate from end users who are potentially spread around the world. These requests will arrive at the closest-to-the-user Google point of presence, where Google Cloud's networking infrastructure looks up the database's location from Firestore's metadata and routes requests to a Firestore Frontend task in the database's region. Outages in this location lookup will affect requests originating locally but destined to databases in any region. This makes it critical for us to target 99.9999% of availability globally for the location lookup service in order to support Firestore availabilities of 99.99% in regional deployment and 99.999% in multi-regional deployment, respectively. This availability is achieved by storing each database's location in a global Spanner database with multiple asynchronously-replicated read replicas. Location lookup uses these replicas, thereby trading off slightly reduced availability of database creation/deletion events with much greater availability of mostly-static location metadata.

B. Billing and Free Quota

A key attraction for developers is Firestore's daily free quota comprising 1 GiB of storage, 50k document reads, and 20k document writes. This lets developers experiment with Firestore and run low-usage services at little or no cost. The free quota is enforced by integration with Google's quota system, operation-based billing is done by logging the count of documents accessed by each RPC and integration with Google's billing system that reads the logs, and storage usage is measured and billed daily; Figure 4 does not show these integrations. The next section discusses Firestore's multi-tenancy, which makes the free quota affordable.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

C. Multi-tenancy and Isolation

Firestore's multi-tenant architecture is key to its serverless scalability. All its components (see Figure 4) and the underlying Spanner database components are shared across large numbers of Firestore databases. As a foundation, all components build on Google's auto-scaling infrastructure [11], so the number of tasks in a given component adjusts in response to load. Thus, idle and mostly-idle databases use extremely few resources, which makes Firestore's free quota and operation-based billing practical. Multi-tenancy however brings isolation challenges: traffic to a single Firestore database can potentially affect the performance and availability of other databases by consuming all or most of the resources in one or more components, or even worse, crashing tasks.

Solving isolation presents several challenges. First, an individual RPC is not a uniform work unit, as its cost can vary significantly—one RPC can cost a million times another—and in ways that are not predictable from RPC inspection, e.g., queries with unknown result-set size. Second, isolation needs to happen on several dimensions: most importantly Firestore Backend CPU and RAM, and Spanner CPU. Finally, database traffic can be very spiky: Firestore requires conforming traffic to grow progressively—increase at most 50% every 5 minutes, starting from a 500 QPS base [12]. Firestore is designed to handle spiky traffic and will still accept traffic that violates this rule as long as it can maintain isolation.

Each component is designed for isolation. For example, we use a fair-CPU-share [13] scheduler in our Backend tasks, keyed by database ID. We also pass the database ID as a key to Spanner's similar fair-CPU-share scheduler. Additionally, certain batch and internal workloads set custom tags on their RPCs, which allow schedulers to prioritize latency-sensitive workloads over such RPCs. We limit the result-set size and the amount of work done for a single RPC, which protects the system against problematic workloads. Firestore APIs support returning partial results for a query as well as resuming a partially-executed query. Most parts of Firestore can split out traffic even on a document granularity to redistribute load. Finally, some components do targeted load-shedding to drop excess work before auto-scaling can take effect; auto-scaling incorporates delays because short-lived traffic spikes do not merit auto-scaling. As discussed in Section VI, it is important to also have manual tools to intervene in emergencies.

D. Writes, Queries and Real-time Queries

A write to the database updates each matching secondary index used by query executions, keeping them strongly consistent with the data, and notifies the Real-time Cache to send notifications to clients with active real-time queries.

1) Spanner Representation: To keep per-database cost low, Firestore maps each database in a region to a specific *directory*[2] within a small number of pre-initialized Spanner databases in that region[3]. Each directory has two tables, `Entities` and `IndexEntries`, which contain the actual Firestore database data.

Each Firestore document is stored as a single row in the (fixed-schema) Spanner `Entities` table. The key-value pairs that constitute a schemaless Firestore document contents are encoded in a protocol buffer [14] stored in a single column, and the Firestore document name (unique key) encoded as a byte-string serves as the key for that row. Spanner provides row-granular atomicity guarantees, which means that the schemaless collection hierarchy in Firestore's data model does not impose any additional locking constraints; two or more Firestore documents can be accessed concurrently independent of their position in the hierarchy.

Each Firestore index entry is stored in an inverted index: a single row in the (fixed-schema) Spanner `IndexEntries` table. The key of this table is an (*index-id*, *values*, *name*) tuple where the *index-id* identifies a particular index for the Firestore database, *values* is the byte-string encoding of the index entry's values, and *name* is the byte-string encoded name of the indexed Firestore document. The encoding of the n-tuple of values in *values* preserves the index's desired sort order. As Spanner tables, like Bigtable [4], support efficient, in-order linear scans by key, a linear scan of a range of `IndexEntries` rows corresponds to a linear scan of a range of the logical Firestore index.

Firestore manages its own indexes and implements its own query engine rather than relying on Spanner's native functionality for two main reasons. First, it is not possible to define a Spanner index that matches Firestore's automatic indexing rules, nor is it possible for such a Spanner index (which has to apply to data from all Firestore databases) to accommodate the per-Firestore-database user-defined indexes and automatic index exclusions. Second, Firestore's query semantics diverge significantly from Spanner's, in particular by allowing sorting on any value including arrays and maps and sorting across fields with inconsistent types, such that a mapping to a Spanner query on the underlying Spanner database is impractical.

Building directly on top of Spanner, with a one-to-one mapping of documents and index entries to Spanner rows, yields significant benefits to Firestore: high availability, data integrity, transactional guarantees, and infinite scaling. In particular, Spanner's automatic load-based splitting and merging of rows into *tablets* [5] (similar to other system's shards or partitions) that hold data for a consecutive key-range allows Firestore to scale to arbitrary read and write loads. Firestore's definition of conforming traffic [12] is designed to conservatively match Spanner's splitting behavior. Firestore's transactions map directly to Spanner transactions, which are lock-based and use two-phase-commits across tablets. Because Spanner uses multi-version concurrency control and assigns *TrueTime* [5] timestamps to transactions, the serializability guarantee on timestamps allows Firestore to perform lock-free

[2]A Spanner concept that guides sharding and placement [5].

[3]Storing each Firestore database in its own Spanner database would require pre-allocating resources for millions of Spanner databases, which is prohibitively expensive with today's state of the art.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

consistent (timestamp-based) reads across a database without blocking writes. The lack of consistency in many queries was a drawback for Datastore's Megastore-based implementation; an important customer mentioned consistency as a reason for migrating from Datastore to Spanner [15].

Adding or removing a Firestore secondary index requires a backfill or backremoval in the Spanner `IndexEntries` table. This is managed by a background service that receives index change requests, scans the `Entities` table for all affected documents, makes the required `IndexEntries` row additions or removals in Spanner, and finally marks the index change as complete. It should be noted that a query that mutates the database also makes all necessary updates to the "IndexEntries" table so that it conforms to an on-going backfill or backremoval.

2) Writes: We describe writes using an example update to the Restaurant application. Say an end-user adds a new rating, which also involves updating the average rating on the restaurant's document. This requires a Firestore transaction that inserts the rating document from Figure 2 and updates the `numRatings` and `avgRatings` fields of the parent restaurant document from Figure 1. The commit of the Firestore transaction is processed by the Backend as follows:

1) Create a Spanner read-write transaction T.
2) In transaction T, read documents `/restaurants/one` and `/restaurants/one/ratings/2` from the Spanner `Entities` table with an exclusive lock[4]. Verify that `/restaurants/one` does exist and that `/restaurants/one/ratings/2` does not exist.
3) Because the request is from a third party, execute the database's *write* security rules (Figure 3) for `/restaurants/one` and `/restaurants/one/ratings/2`. Add the update of row `/restaurants/one` and the insert of row `/restaurants/one/ratings/2` in `Entities` to transaction T.
4) Use the (cached) index definitions to compute the index entry changes for the two documents. The result is the removal of the old index entries for `numRatings` and `avgRatings` for `/restaurants/one` and additions of new ones for their new value, and addition of new index entries for all the fields of `/restaurants/one/ratings/2`. Add the corresponding row insertions and deletions in `IndexEntries` to transaction T, thereby ensuring Firestore indexes stay strongly consistent with the documents.
5) Pick a reasonable max commit timestamp M and start a two-phase-commit with the Real-time Cache by sending one or more `Prepare` RPCs with max commit timestamp M. The results of each RPC contain a minimum allowed commit timestamp m_i; Section IV-D4 describes which tasks in the Real-time Cache the Backend communicates with.
6) Commit Spanner transaction T with minimum allowed timestamp $max(m_i)$ and maximum allowed timestamp M: Spanner acquires additional exclusive locks on the specific `IndexEntries` rows, and then atomically commits the changes to `Entities` and `IndexEntries`. This may involve updates across multiple Spanner tablets and servers. The lock acquisitions at this stage can conflict only with the read-locks from queries executing within a transaction[5], as `IndexEntries` rows include the unique document name and the document is already under an exclusive lock.
7) Finish the two-phase-commit with the Real-time Cache by sending corresponding `Accept` RPCs with the outcome of the Spanner commit; at this point the Real-time Cache should have the name of each deleted document, a full copy of each inserted document, and a full copy of each modified document together with the exact changes. The Real-time Cache tracks these mutations in memory sorted in timestamp-order.

[4]Sub-document granular locking is not supported because a well-designed data model has many small documents, and sub-document concurrency is unnecessary.

There are multiple points where this process can fail, with varying consequences:

- `/restaurants/one` does not exist, `/restaurants/one/ratings/2` already exists, or the security rules deny the request: an error is returned to the user.
- The `Prepare` RPC fails because the Real-time Cache is unavailable (this should be rare): the write fails and an error is returned to the user.
- The Spanner commit definitively fails, e.g., due to contention or not being able to respect the maximum timestamp. The `Accept` RPC notifies the Real-time Cache, and an error is returned to the user.
- The Spanner commit has an unknown outcome, e.g., it times out. The `Accept` RPC notifies the Real-time Cache that the write outcome is unknown, which in turn discards the in-memory sequence of mutations.
- The Spanner commit is successful but the `Accept` RPC is not received by the Real-time Cache. The in-memory sequence of mutations is (eventually) discarded by the Real-time Cache, but the write is acknowledged to the end-user.

Insertion of documents with many fields results in a larger number of index entries that need to be added, and that translates to a Spanner transaction potentially across more tablets, which can impact commit latency. Indexing a field that increases sequentially, e.g., a document creation timestamp, implies the insertion of consecutive rows in the "IndexEntries" table as documents get created. This workload is inherently difficult to split.

Network latency between replicas is higher for a multi-regional deployment, and Spanner needs a quorum of replicas to agree before committing a write, leading to higher Firestore

[5]As discussed in subsubsection IV-D3, a timestamp-based query runs without locks.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

write latency in multi-regional deployments than in regional ones.

Spanner also has a transactional messaging system that allows its user to persist information that can be used to perform asynchronous work. This system is used by the Firestore Backend to implement write triggers (subsection III-F). If an incoming request matches a trigger, the Backend persists a message with the changes to document(s), which is then asynchronously removed and delivered to the Cloud Functions service to execute the specified handler.

3) Queries: Firestore's query engine executes all queries using either a linear scan over a range of a single secondary index in the Spanner `IndexEntries` table, or a join of several such secondary indexes, followed by lookup of the corresponding documents in the `Entities` table, with **no** in-memory sorting, filtering, etc. For instance,

```
select * from /restaurants
where city="SF" and type="BBQ"
order by avgRating desc
```

is satisfied by the secondary index (city **asc**, type **asc**, avgRating **desc**). Firestore's automatically defined single-field indexes support simple queries, such as

```
select * from /restaurants
where city="SF" limit 10

select * from /restaurants
where numRatings > 2

select * from /restaurants
order by avgRating desc
```

To reduce the need for user-defined indexes, Firestore joins existing indexes. Thus, a query like

```
select * from /restaurants
where city = "SF" and type = "BBQ"
```

is executed by joining automatic single-field index (city **asc**) with (type **asc**), and a query like

```
select * from /restaurants
where city="New␣York" and type="BBQ"
order by avgRating desc
```

by joining user-defined indexes on (city **asc**, avgRating **desc**) and (type **asc**, avgRating **desc**). Selecting the ideal set of indexes to join for a query is intractable, so Firestore's query engine uses a greedy index-set selection algorithm that optimizes for the number of selected indexes. If no such set exists, Firestore returns an error message that includes a link for adding the required index via the Google Cloud Console. In practice, these error messages let developers add the required indexes during testing. We do occasionally receive support cases for query performance caused by slow index joins that are remediated by defining additional indexes.

The execution of a non-real-time query starts by verifying the security rules for the collection specified in the query.

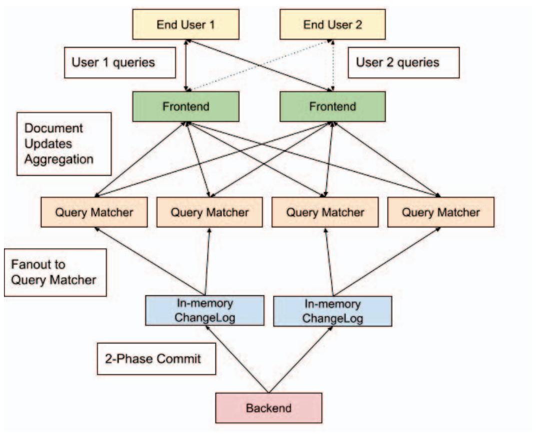


Fig. 5: A more detailed diagram of the Real-time Cache.

The query planner then uses the (cached) index definitions to pick the best index(es) for the query. The Backend reads, and, if necessary "zig-zag joins" [16], the row ranges from `IndexEntries`, then fetches the corresponding rows from `Entities`, and returns the documents.

Queries can be executed within a Firestore transaction (the Spanner-level reads of `IndexEntries` and `Entities` acquire read locks, guaranteeing consistency with other transactions) or outside a transaction using Spanner's lock-free consistent, timestamp-based reads. In the former mode, long-lived or large transactions may lead to lock contention and deadlocks that are resolved by failing and retrying such transactions. A timestamp-based query suffers no such problem.

4) Real-time Queries: A client registers one or more real-time queries via a long-lived connection with a Frontend task. The Frontend task uses this connection to deliver updates to the result sets of those queries. The updates are delivered to applications as incremental, timestamped snapshots, comprised of a delta of documents added, deleted, and modified from the prior snapshot. This section provides only a high-level overview of this system. A more detailed description of this infrastructure is outside the scope of this paper.

As Figure 5 shows, the Real-time Cache comprises 2 components—the In-memory Changelog and the Query Matcher. A separate mechanism establishes and shares consistent ownership of document-name ranges to specific Changelog and Query Matcher tasks.

The request/response flow for a real-time query is as follows:

1) A client creates or reuses a long-lived connection to a Frontend task and registers a new real-time query.
2) The Frontend task creates state for this real-time query and then obtains the query's initial snapshot by forwarding the query to a Backend (which runs it like any other query) to retrieve all the documents matching the query; the response includes the corresponding Spanner timestamp of that data, which we will call max-commit-version.
3) The Frontend task sends the client an initial snapshot

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

based on the Backend's response and records max-commit-version with the query state. All subsequent updates to this result set will be delivered to the client as incremental snapshots.

4) The Frontend task then sends one or more `Subscribe` RPCs to Query Matcher tasks that own the specific document-name ranges that cover the query's result set. The `Subscribe` RPC includes the query and the max-commit-version. This informs the Query Matcher task to register the query for matching, and to subsequently send only the document updates that match the query and that have a Spanner commit timestamp later than the max-commit-version.
5) A Changelog task forwards document updates (received via `Accept` RPCs from the Backend) to the Query Matcher task owning the corresponding document-name range. On receiving the document, the Query Matcher matches it with all the queries registered for that key range and sends the matched documents to the Frontend task.
6) Because updates for a single query come from the multiple Query Matcher tasks to which the query was subscribed, the Frontend task is responsible for tracking when it has received all the updates necessary to reach a consistent timestamp. Only then does it send the accumulated delta of those updates to the client as a new incremental snapshot for that query's result set; it also then updates the query's max-commit-version to that latest timestamp. Changelog tasks generate a heartbeat every few milliseconds for every idle key range; this heartbeat is crucial for the Frontend tasks to know that they have received all updates when a document-name range is otherwise idle.

A client can open many real-time queries to the same database, multiplexed over the same long-lived connection to the Frontend task. Updating these queries to inconsistent timestamps would be confusing to the end-user where the results from multiple queries may be presented together. To avoid this, queries on the same connection are only updated to a timestamp t once all queries' max-commit-version has reached at least t.

When a Changelog task receives a `Prepare` with max timestamp M, it responds with a minimum timestamp m. The maximum timestamp (plus a small margin) sets how long the Changelog will wait for the corresponding `Accept`. The Changelog knows it has a complete sequence of updates until time t once it has received `Accept` responses for all `Prepare` RPCs that it sent out with a minimum timestamp less than t. This machinery, and the timestamp processing in the Frontends, relies crucially on the globally-consistent, causally-ordered timestamps provided by Spanner [5].

The `Accept` indicates whether the write was successful, failed or had an unknown outcome. If successful, the Changelog tasks forwards the associated document updates to the Query Matcher task (that owns the name range) with the commit timestamp; a failed write is dropped. There are several error scenarios that may occur, but we discuss only one given the lack of space: if the Changelog times out while waiting for an `Accept` or the `Accept` indicates an unknown outcome, the system cannot guarantee ordering of the updates for that name range. Then, the Changelog task marks that name range as out-of-sync and signals that all the way up to all Frontend tasks with a real-time query that matches the name range. The Frontend task then aborts all accumulated state for that query and redoes the steps starting with the initial query request to the Backend. This reset is fast, and is mostly transparent to the end-user of the application. In general, this recovery method is a fail-safe mechanism to handle difficult error conditions, such as the crash/restart of a particular task. Load-balancing is achieved by dynamically changing the document-name range ownership across Changelog and Query Matcher tasks by leveraging the Slicer [17] auto-sharding framework.

E. Disconnected Operation

The Client (Mobile and Web) SDKs build a local cache of the documents accessed by the client together with the necessary local indexes. It uses the local cache to provide low latency responses to client queries without the network penalty. Mutations to documents by the client are acknowledged immediately after updating the local cache; the updates are also flushed to the Firestore API asynchronously.

The local cache is updated whenever it receives notifications over the long-lived connection it maintains with a Frontend task. A disconnected client can therefore continue to serve queries and updates using its local cache, and reconcile its local cache when it eventually reconnects with Firestore. The Client SDK is also responsible for guaranteeing consistency across the multiple real-time queries a client may have active.

Based on their privacy preferences, an end user can choose to persist their local cache. This choice affects the behavior after a device is restarted; persistence provides a warm cache as a starting point for requests to the Client SDK.

It should be noted that the customer is not billed for any work that can be satisfied by the local cache; only the traffic to/from the Firestore service is billed as described in subsection IV-B.

V. Evaluation

We share some production data, show latency variance with change in important parameters, evaluate one isolation mechanism, and analyze factors that make Firestore easy to use. All data in this section is presented as relative to a median or as comparisons across the changing x-axis parameter.

A. Production Statistics

The four million Firestore production databases accessed by over a billion end-users each month are evidence of Firestore's wide adoption, ease of use and scalability. The scalability of Firestore is also demonstrated by the variability in usage patterns seen across customers, all of whom interact with the same multi-tenant Firestore tasks and Spanner databases. We present this variance as boxplots [18] in Figure 6 using values

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

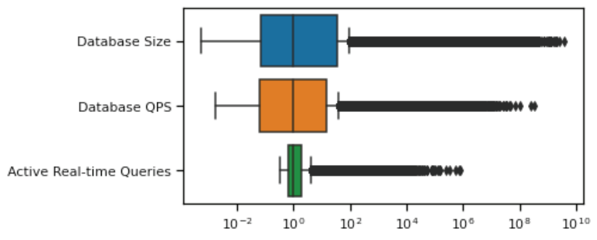


Fig. 6: Various database properties normalized by median.

normalized to their respective median. This data is taken across all Firestore databases that have seen activity in the last 28 days; it does not include the small number of databases on Megastore that remain to be migrated.

The first boxplot shows that some Firestore databases differ from the median storage size by more than nine orders of magnitude, and that is several orders of magnitude larger than the amount of storage that a single machine can provide. As described earlier, Spanner splits and merges data stored in its tablets across thousands of servers [5], and therefore seamlessly handles all sizes of Firestore databases. Thus, even a petabyte-scale Firestore database can expect no negative impact on availability or performance.

The variance in throughput (QPS) across databases likewise demonstrates how Firestore handles scaling by 9 orders of magnitude from the median. Automatic scaling, Spanner load splitting, and fair scheduling in the multi-tenant Firestore architecture are critical to ensuring that the latency and availability of our databases are independent of the load.

The number of active real-time queries is also highly variable, with some databases seeing several hundred thousand times the number of active queries as the median. We also find many instances daily where the active query count for a given database grows twenty-fold within minutes. As described earlier, our architecture allows Firestore to independently scale the Real-time Cache components, allowing even a single change to update millions of end-user devices.

B. Latency

We evaluate Firestore's scalability using synthetic benchmarks run against a production Firestore database in the nam5 multi-region [19] located in the central US. All benchmarks use strongly consistent reads; that choice together with the multi-regional deployment ensures the experiment tests worst-case performance. The y-axis of all graphs in this subsection start from zero and use linear scales. However, the reader should assume that the y-axis of separate figures are to different linear scales unless stated otherwise.

1) Scalability: We ran the YCSB [20] benchmark: workload A with 50% reads and 50% updates and workload B with 95% reads and 5% updates. We used a uniform key distribution with 900-byte sized documents, each composed of a single field of that size. Tests were run for 10 minutes for each target QPS throughput; the data shown is based on measuring the last 5 minutes to allow the system to stabilize.

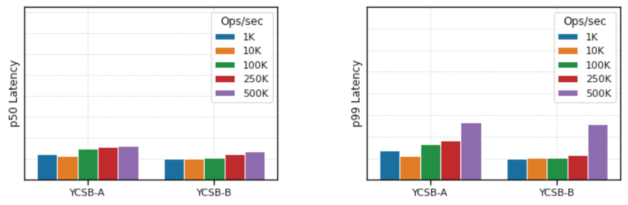


Fig. 7: Read latency for both YCSB workloads.

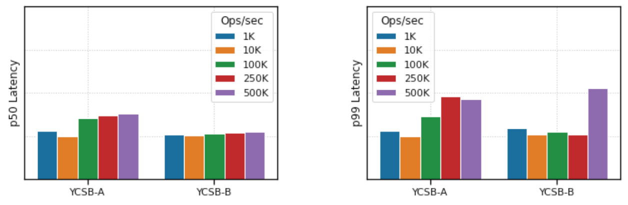


Fig. 8: Update latency for both YCSB workloads.

Figure 7 and Figure 8 show read and update latencies from both workloads, respectively. The y-axes in all four graphs use the same linear scale and are therefore comparable. As we know, writes are more demanding to process than reads, so workload B generally sees lower latencies than workload A as A has a larger proportion of writes. The p50 read and update latencies remain roughly constant across throughput levels for both workloads. There is an increase in p99 latency at higher QPS, mainly on the write-heavy workload A. This is partly because YCSB workloads ramp up very rapidly. Given Firestore's serverless nature, capacity is not pre-allocated for individual databases, and scale-up instead relies on autoscaling of Google infrastructure to create more Backend tasks to share the load and dynamic load splitting in Spanner, and this particularly affects writes. We observed a decrease in p99 latency during the 10 minute period of both workloads, especially in the higher QPS tests, down to match latency of the lower QPS tests. Firestore best practices recommend steady exponential ramp-up [12].

Another common scenario in Firestore is the broadcasting of query updates by the Real-time Cache to multiple end-users; one example is when end users running an application that displays sporting-event scores receive a query update due to a team scoring. To test this scenario, we set up a workload that writes to a single document once every second, while an increasing number of Firestore clients open a real-time query that includes that document in its result set. Thus, each write to the document triggers a small update that is sent to each client. We report the notification latency, measured as the delay from when the Firestore Backend receives an acknowledgement from Spanner denoting a write is committed until the corresponding notification is sent to all clients by the Frontend. Figure 9 shows that notification latency remains relatively stable even with an exponential increase in the number of Listen connections. The increase in active real-time queries increases the load on Frontend tasks, which leads autoscaling [11] to quickly scale up the number of Frontend tasks, independently of the rest of the system. Although not shown

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

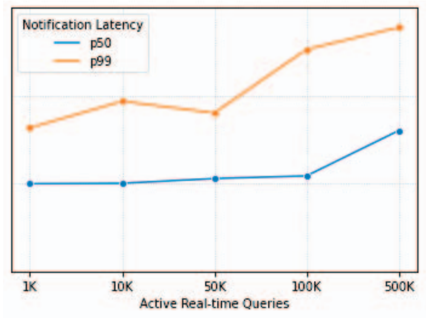
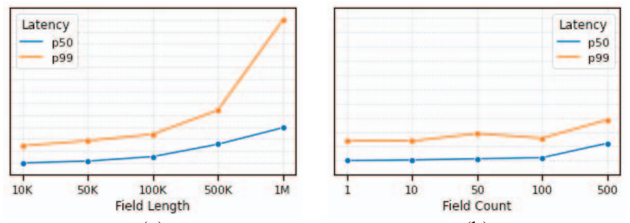


Fig. 9: Notification latency on a linear y-axis with increase in number of client connections.



(a) (b)

Fig. 10: Latency of document commits with increasing (a) size, via field length (b) index count, via field count. The y axes on both these graphs share the same linear scale, and are therefore directly comparable.

in this graph, we also observe that the commit latency remains constant throughout the experiment because of this separation.

2) Data Shape: Two obvious properties affecting latency of Firestore writes are the size of documents being committed as well as the number of indexes being updated. The latter is particularly relevant because—unless specifically exempted—Firestore automatically indexes all fields for easy querying.

To illustrate these relationships, we ran two experiments with 10 QPS of Firestore commits, where each commit adds a single document. In the first experiment, each document comprises a single field with a varying length of single-byte characters, from 10KB to almost 1MiB, which is the maximum document size supported by Firestore. In the second experiment, each document has a varying number of numeric-value fields from 1 to 500, which results in a linear increase in the number of index entries written per commit. From a performance standpoint, for the same number of index entries, there is no significant performance difference between one large field and many small fields of the same total size, or an array or map with many elements again of the same total size: Firestore indexing flattens out fields such as arrays or maps to index each element, and therefore the corresponding performance is similar to that of a document with as many fields. The experiment was preceded by initializing the database with enough data to ensure that commits spanned multiple tablets and thus adding a single document required a distributed Spanner commit. Each data point shows latencies from a 10 minute measurement interval.

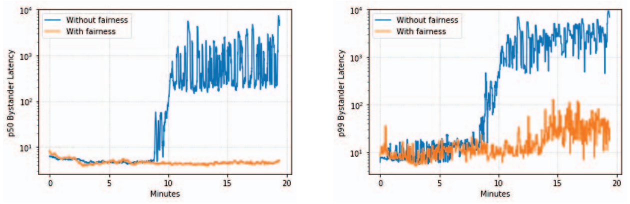


Fig. 11: Query latency of "bystander" database with and without fair CPU scheduling.

Figure 10a shows that the increase in latency has a base cost plus a linear, size-dependent cost (note that the x-axis is not linear). Figure 10b demonstrates that index entry count does not significantly impact latency until beyond several hundred fields. It shows that commit latency increases more slowly with the increase in number of rows—each index entry is an additional Spanner row written to the `IndexEntries` table. In summary, commit latencies are dominated by document and field sizes and not by field count.

C. Isolation

Subsection IV-C outlines several isolation mechanisms used to make Firestore's multi-tenancy feasible. One crucial isolation feature is fair CPU scheduling in the Firestore Backend, which prevents a single database's requests from starving other databases of CPU when requests rise quicker than automatic scaling can react. We evaluate this isolation with a small scale, fixed capacity (no automatic scaling) Firestore environment with fair CPU scheduling enabled or disabled. We send two workloads to this environment: a "culprit" database sends CPU-intensive (due to an inefficient indexing setup) queries that linearly ramp up to 500 QPS to hit scaling limits of the test environment, and a "bystander" database sends 100QPS of single-document fetches. As Figure 11 shows, when capacity limits are reached halfway through the experiment, a lack of CPU fairness leads to a significant degradation of the bystander database's latency. The fair scheduling keeps latency impact to a minimum, leaving only a small increase in p99 latency (note the log scale). Firestore's production environment robustly handles all types of traffic spikes thanks to this isolation feature and many others, such as automatic scaling.

D. Ease of Use

Outside of the quarter million developers actively building on Firestore, there is no convenient quantitative way to showcase Firestore's ease of use. Instead, we discuss some aspects of the ease with which Firestore can be used to build a sample application and the corresponding number of lines of Javascript code. This example—illustrated as a step-by-step guide by the Firestore Web Codelab [6]—is a functional restaurant recommendation web application, which lets viewers see a list of restaurants with filtering and sorting, and view and add reviews.

The initialization steps—creating and enabling a database, setting up rudimentary access control, and a web server—are all accomplished by running a few commands. The onSnapshot() method is used to listen to the results of a query. The

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

developer specifies a callback that receives the initial query result snapshot. Subsequently, each time the result set changes, another call updates the snapshot. The developer can also view the changes to the result set, i.e., the documents added, deleted or modified. The result set state is populated or updated from the local cache and from updates sent by the Real-time Cache; this is all handled seamlessly by the Client SDKs.

This application is 841 lines of Javascript code, of which 92 are to access Firestore. However, this Codelab includes creation of sample restaurant data and other hardwired constants, without which the application shrinks to 572 lines, of which 66 are for reading from and writing to the Firestore database. Custom security rules (25 lines) allow reads to authenticated users and creation of new reviews as long as the review contains the authenticated user's unique identifier.

We also implemented a "Todo List" application that lets users share a todo list, add new items, mark items as done, and delete them. This requires 112 lines of Javascript code, of which 37 are for Firestore database access. Security rules (8 lines) allow everyone to read and create todo items, but only an item creator can mark an item as done or delete it.

BeReal is an example of a recently viral social application that was written using Firestore and where its ease of use has been highlighted as an advantage [21].

VI. Lessons in Practice

We present a selection of the many practical lessons learnt from running Datastore and Firestore over more than a decade.

Backwards compatibility is essential for a cloud service that is constantly updated under live traffic. The default guarantee for updating Firestore is bug-for-bug API compatibility. Very occasional, small behavior changes are possible but only with strong motivation. When necessary, such changes are preceded by a comprehensive investigation—examples are analysis of all RPCs seen during a sufficiently long period or a scan of the entire corpus of customer data—to identify all customers that may be impacted by the change, and working directly with them to address potential risks. We twice rewrote the Firestore query planner. These rewrites were extensively tested with A/B comparison of query execution to confirm zero customer impact before rollout. The migration of Datastore from the Megastore-backed to a Spanner-backed system unavoidably reduced maximum key size, affecting a tiny number of documents for very few customers. We contacted these customers directly to ensure their data was handled correctly.

Relatedly, Firestore is not versioned, so rollouts directly and immediately impact customer traffic. Avoiding problems from these regular (weekly) rollouts requires detecting problems early and rolling back the release, if necessary. To achieve this, we rely on Google's internal rollout principles—slow code and configuration rollouts within a region and gradual rollout across regions. We use automated A/B testing for some number of minutes of the current against the new release across many metrics before the rollout is allowed to proceed. This is done at sub-region granularity because traffic patterns can differ significantly across regions.

Data integrity is a core requirement of any database. We rely both on Spanner's data integrity guarantees for data at rest, and periodic data validation jobs at both the Spanner and Firestore layers to verify the correctness of data and consistency of indexes. However, mass-produced machines themselves are unreliable [22], [23] and may corrupt in-memory data. We are actively addressing these issues through the addition of end-to-end checksums to protect in-flight RPCs.

As discussed earlier, isolation is hard, and our techniques for maintaining isolation are not always guaranteed to work, especially when confronted with sudden traffic spikes of unusual workloads that, e.g., require much more RAM than the typical RPC. We use two tools to quickly mitigate such challenges. One is a low-tech manual tool that limits the number of per-task in-flight RPCs for a given database, which has been one of our more effective mechanisms for preventing isolation failure while waiting for fixes (capacity changes or code updates) to rollout. This tool may not suffice in some cases, such as traffic triggering a bug that leads to task crashes or when limiting a customer's workload is highly undesirable. In such cases, all traffic for that database can be routed to a separate pool (of tasks) for the impacted component, thereby isolating it completely. This pool can also be configured to auto-scale to the database's traffic.

Finally, the design of the Firestore API was informed by many lessons learned from Datastore; we present one significant lesson. Our customers found the Megastore-based implementation restrictive for organizing their data, which had to be carefully organized into *entity groups* to support transactional updates and strongly-consistent queries. Furthermore, the write throughput to each entity group was limited. This forced most customers into using eventually-consistent queries [15]. Firestore leverages Spanner to remove all of these limitations, allowing unrestricted transactions and strongly-consistent queries, with no transaction-rate limits.

VII. Related Work

Lotus Notes was the first document-oriented system, and was emulated later by what came to be known as NoSQL document stores. When it launched in 1989, Notes had many advanced features that did not then exist in RDBMSs—e.g., disconnected operation with replicated documents in a client-server environment, field-level authorization and encryption, n-gram indexing of strings enabling fuzzy searching, form-based workflows, triggers, and tree of trees indexes to support nested views with sophisticated collation options. Because Notes was aimed at small workgroup environments, it hit scaling and transactional problems when used in large enterprises as a mail system and a document store. More than a decade later, these issues were addressed with the introduction of log-based recovery and by allowing a single database to span more than one file [24]. Notes preceded the cloud and the big data revolution, so was designed for on-premise database sizes.

Microsoft's Azure Cosmos DB (originally named Azure DocumentDB [25]) is a modern day cloud-scale document store, which is similar to Firestore. While Cosmos DB uses

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

the file system directly to persist its documents, Firestore relies on the scaling and decade-long production quality of Spanner [5]. We have discussed earlier how Firestore uses Spanner's data modeling constructs in a stylized way to store document collections and their associated indexes. It should be noted that this results in what could be characterized as unnormalized tables due to the redundant storage of collection name in every Spanner row representing a document, and *index ID* in every Spanner row representing an index entry; this is necessitated by our multi-tenant layout in Spanner. Cosmos DB supports a SQL dialect and JavaScript to query, using relational or hierarchical constructs, JSON documents with tunable transaction consistency levels. It also supports stored procedures and triggers, while Firestore supports triggers by integrating with Google Cloud Functions. One of Firestore's main goals is ease of mobile-friendly application development with an easy-to-understand billing model and disconnected operation for mobile clients with automatic reconciliation of parallel updates.

Amazon DynamoDB [26], [27] has its own storage engine that scales, whereas Firestore leverages Spanner. The evolution of DynamoDB from SimpleDB also differs from the evolution of Firestore from Datastore. One unique feature of DynamoDB is that the system continuously and proactively monitors availability both on the server side and on the client side. In addition to data replicas, it also supports log replicas to improve availability. It supports strongly and eventually consistent reads. For efficient metadata management, it relies on an in-memory distributed datastore called MemDS.

The original version of MongoDB did not have transactional guarantees. Now, it stores all data in BSON format (binary form of JSON), using the WiredTiger storage engine which supports transactions and tunable levels of consistency by exposing what are called writeConcern and readConcern levels that are usable with each database operation [28]. This aspect of MongoDB has necessitated the invention of a speculative execution model and data rollback algorithms. MongoDB does support secondary indexes, an ad hoc query language, complex aggregations, sharding, etc. The adaptation of the TPC-C benchmark to the document database model of MongoDB is presented in [29]. MongoDB includes Change Streams [30], which allows interested parties to subscribe to changes in one or all collections in a database. This differs from Firestore's real-time query feature, which supports listening on complex queries such that only relevant changes are streamed to the listener, and scaling to arbitrary numbers of listeners.

Couchbase [31] is a document database system that supports JSON as its data model. Based on a shared-nothing architecture, Couchbase supports indexes and declarative querying using SQL-like queries, including joins, aggregations, and nested objects. As an alternative to the original Couchstore storage engine, a new storage engine called Magma has been developed recently [32] to support write-intensive workloads. Its goal is to improve on write and space amplification.

There is a large body of work on standing or continuous queries for relational database systems, but we are unaware of work that is comparable to Firestore's real-time queries in other document database systems.

VIII. Future Work

The Firestore query API was designed to be simple for application developers, but it may become limiting as an application matures. We are working on adding query functionality while conforming to the design parameters of predictable query scaling and efficient real-time updates. These changes will require extending our billing model that is currently based on only the number of documents in the result set; a COUNT query returns a single value but may count millions of documents, in-memory filtering may require examining and discarding many documents. However, such extensions cannot break the pay-as-you-go billing that is essential to the serverless experience.

It would be beneficial to push down more of Firestore query evaluation into Spanner for increased efficiency and reduced latency. However, translation of a Firestore query into one on the underlying Spanner schema (that represents Firestore's data) may produce a sufficiently complex query that Spanner's query planner cannot execute it efficiently. Unlocking this problem is an active area of investigation.

Some customers wish to add light schema restrictions on their previously schemaless data in some mature applications. Providing opt-in schema functionality will allow mapping those fields directly into our underlying Spanner schema, unlocking the aforementioned push-down efficiency and potentially more query functionality.

We are exploring improved isolation by selective slowdown or rejection of traffic of a given database when under memory pressure, based on the memory consumed by in-flight queries to that database. So far, we have focused on isolation mostly between databases, but Firestore customers need isolation also within their database: for example, a bug in their daily batch job should not lead to rejection of user-facing traffic. Adding support for intra-database isolation will require API-level changes, e.g., in the form of QoS indications.

IX. Conclusion

In this paper, we presented and analyzed the building blocks of Firestore that are key to its popularity with the application developer community. We showed how its schemaless data model, serverless operations, and simple billing model combined with the Firebase SDKs, provides a convenient ecosystem with a low barrier of entry to developers to rapidly prototype, deploy, iterate, and sustain applications. We showed how Spanner and Google infrastructure were leveraged to allow QPS and storage scaling, and presented an overview of how the Real-time Cache and client SDKs provide a seamless experience of real-time notifications to clients even in the presence of network connectivity issues.

We thank the many Googlers and ex-Googlers who have worked to make Datastore and Firestore a success. In particular, Datastore development was successively led by Ryan Barrett, Max Ross and Alfred Fuller. The Firestore API development was led by Alfred Fuller and Andrew Lee.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.

References

[1] A. Arasu, S. Babu, and J. Widom, "CQL: A language for continuous queries over streams and relations," in *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, ser. Lecture Notes in Computer Science, G. Lausen and D. Suciu, Eds., vol. 2921. Springer, 2003, pp. 1–19. [Online]. Available: https://doi.org/10.1007/978-3-540-24607-7_1

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

[3] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "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, 2011, pp. 223–234. [Online]. Available: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[6] "Cloud Firestore Web Codelab," https://firebase.google.com/codelabs/firestore-web#0, accessed 2022-10-18.

[7] "Firebase Authentication," https://firebase.google.com/docs/auth, accessed 2022-10-18.

[8] "Firebase Security Rules," https://firebase.google.com/docs/rules, accessed 2022-10-18.

[9] "Google Cloud Functions," https://cloud.google.com/functions, accessed 2022-10-18.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[11] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[12] "Firestore: Ramping up traffic," https://firebase.google.com/docs/firestore/best-practices#ramping_up_traffic, accessed 2022-10-18.

[13] C. A. Waldspurger and W. E. Weihl, *Stride scheduling: Deterministic proportional share resource management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

[14] "Protocol buffers," https://developers.google.com/protocol-buffers, accessed 2022-10-18.

[15] "How Pokémon GO scales to millions of requests?" https://cloud.google.com/blog/topics/developers-practitioners/how-pok%C3%A9mon-go-scales-millions-requests, accessed 2022-10-18.

[16] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 3, pp. 239–264, 1986.

[17] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, A. Shraer, A. Merchant, and K. Lev-Ari, "Slicer: Auto-sharding for datacenter applications," in *OSDI'16: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, 2016, pp. 739–753.

[18] "Box plot," https://en.wikipedia.org/wiki/Box_plot, accessed 2022-10-18.

[19] "Cloud Firestore locations," https://firebase.google.com/docs/firestore/locations#location-mr, accessed 2022-10-18.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[21] "BeReal builds a real and authentic social media platform on Google Cloud," https://cloud.google.com/blog/topics/startups/bereal-creates-reality-based-social-media-using-google-cloud, accessed 2022-10-18.

[22] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[23] "Silent Data Corruption at Scale," https://www.sigarch.org/silent-data-corruption-at-scale/, accessed 2022-10-18.

[24] C. Mohan, R. Barber, S. Watts, A. Somani, and M. Zaharioudakis, "Evolution of groupware for business applications: A database perspective on lotus domino/notes," in *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, Eds. Morgan Kaufmann, 2000, pp. 684–687. [Online]. Available: http://www.vldb.org/conf/2000/P684.pdf

[25] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra *et al.*, "Schema-agnostic indexing with azure documentdb," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1668–1679, 2015.

[26] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 729–730.

[27] S. Perianayagam, A. Vig, D. Terry, S. Sivasubramanian, J. C. Sorenson III, A. Mritunjai, J. Idziorek, N. Gallagher, M. Elhemali, N. Gordon *et al.*, "Amazon dynamodb: A scalable, predictably performant, and fully managed nosql database service," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 1037–1048.

[28] W. Schultz, T. Avitabile, and A. Cabral, "Tunable consistency in mongodb," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2071–2081, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2071-schultz.pdf

[29] A. Kamsky, "Adapting TPC-C benchmark to measure performance of multi-document transactions in mongodb," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2254–2262, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2254-kamsky.pdf

[30] "How Do Change Streams Work in MongoDB?" https://www.mongodb.com/basics/change-streams, accessed 2022-10-18.

[31] M. A. Hubail, A. Alsuliman, M. Blow, M. J. Carey, D. Lychagin, I. Maxon, and T. Westmann, "Couchbase analytics: Noetl for scalable nosql data analysis," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2275–2286, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2275-hubail.pdf

[32] S. Lakshman, A. Gupta, R. Suri, S. D. Lashley, J. Liang, S. Duvuru, and R. Mayuram, "Magma: A high data density storage engine used in couchbase," *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3496–3508, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3496-lakshman.pdf

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on September 11,2024 at 15:28:31 UTC from IEEE Xplore. Restrictions apply.