The Live Database: Firestore's Scalable and Consistent Realtime Queries

Matt Bell
Google
United States
mbell@google.com

David Gay
Google
United States
dgay@google.com

Per Jacobsson
Google
United States
pjacobsson@google.com

Ram Kesavan
University of Illinois
Urbana-Champaign, United States
ram.kesavan@gmail.com

Namita Lal
Google
United States
nll@google.com

Xijiao Li
Google
United States
xijiaoli@google.com

Hui Wu
Google
Canada
wuandy@google.com

ABSTRACT

Google's Firestore is a NoSQL serverless database service that scales to millions of queries per second and petabytes of storage, serving over 8 million databases and 1.5 billion monthly active end users. A key reason for its popularity is its unique realtime query feature, which greatly simplifies web and mobile application development. Realtime queries allow an application to present an up-to-date and strongly-consistent view of the state of the database to the end-user, and to receive fast notifications of changes to the database state.

In this paper we present the architecture of the Firestore realtime stack, how it provides seamless scaling and high availability while supplying strongly-consistent responses to clients, including evaluations simulating workloads that represent large applications. We also explain how the Firestore SDK enables the application to deliver a smooth user experience through periods of network disconnection.

PVLDB Reference Format:
Matt Bell, David Gay, Per Jacobsson, Ram Kesavan, Namita Lal, Xijiao Li, and Hui Wu. The Live Database: Firestore's Scalable and Consistent Realtime Queries. PVLDB, 19(12): 4130 - 4142, 2026.
doi:10.14778/3827998.3828021

1 INTRODUCTION

Many mobile and web applications, such as social media, chat, and games, have a similar set of requirements. They are typically interactive and need automatic and realtime updates to relevant data. They need to support concurrent access by a large number of simultaneous users and high availability with automatic and transparent failover. Smooth "offline" behavior — eg., graceful degradation when a client loses network connectivity and automatic reconciliation when connectivity is restored instead of freezing — is also critical to better application experience.

Traditionally a complete solution to these problems involves the application developer setting up complex infrastructure (application servers, sharded databases, streaming protocols, load balancing and autoscalers, etc). Firestore is a NoSQL cloud database whose feature set significantly simplifies this challenge, allowing applications to connect to the database directly from an end user's device and keep their data continuously synchronized with minimal lag using realtime queries. Realtime queries give the application developer an easy-to-use programming model which lets their application "listen" to relevant changes in the database, using a database query API.

Consider for instance this paper's running example, a global chat application, with individual chat rooms that users can create, join and message. The database stores the set of rooms, and for each room, all messages. The chat application will have one realtime query tracking the set of available chat rooms, and one realtime query for each room the user has joined that tracks the messages in that room. With these queries, the application is continuously aware of the set of available rooms, and immediately notified on all new messages in joined rooms.

Our initial Firestore paper [22] gives an overall presentation of all components of Firestore, including a brief description and evaluation of Firestore's realtime query implementation. Here, we expand on that description and evaluation, focusing in particular on how Firestore achieves the following hard-to-reconcile requirements.

Low latency: Applications in the "chat room" space require that changes issued from one device be made visible on another device querying the same data with low latency – O(100ms). The paper explains the techniques used in Firestore, like efficient query matching and the streaming protocol, that power low latency notifications.

Automatic scalability and high availability: Firestore's automatic scalability allows a database to scale from zero to serving

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828021

millions of active users issuing realtime queries in the face of hundreds of thousands of writes per second, without requiring the developer to explicitly design their application to shard or partition the data. The system also provides high availability guarantees, which includes performance isolation that absorbs spiky and noisy neighbor behavior in its multi-tenant architecture. We describe how Firestore allows efficient scalability in both read- and write-dimensions.

Strong consistency: The system must be able to provide strong guarantees that the data replicated to the local device and seen by the user is consistent with what is stored in the database. Similarly, when an application opens multiple realtime queries, the result sets of those queries presented to the user should be from a consistent snapshot or timestamp of the database. For example, one must not create (and join) a new chat room, and receive messages from that room without also seeing the room in the set of available rooms. We describe the design of the flow of data through the system and how mutation and time messages are used to build a consistent snapshot for the active realtime queries, while not impeding the requirements of low latency or automatic scalability.

Offline mode: Allowing an end user to transparently keep using an application even if there is no connection to the database is a common scenario with mobile applications. If the user loses network connectivity, they should still be able to send new messages or even create and join a new chat room, and that room should immediately show up in the available chat rooms query's results. Firestore supports disconnected operations and automatic reconciliation; in particular we describe how we optimize reconciliation after long disconnected periods, and leverage the same support to allow local updates to be immediately reflected in active queries and eventually reconciled when the updates reach the database, which improves the interactive response of applications.

The design of Firestore's realtime query support was inspired by Firebase's Realtime Database, which Google acquired in 2014. Work on Firestore started immediately thereafter, with an alpha release in 2016 and general availability in January 2019. The realtime query system is built on top of well-known Google infrastructure including Spanner [13], TrueTime [13], Chubby [8], and Slicer [3] – all critical dependencies that make this design practical. The paper focuses both on new components and how existing systems are orchestrated to support the realtime query use case.

To the best of our knowledge, this is the first paper that presents the design and explains the inner workings of a production-grade database system that provides automatically scalable, highly available, strongly consistent and low latency realtime queries.

2 FIRESTORE DATA MODEL AND QUERIES

Firestore is a document-oriented database with support for ACID transactions. Documents are key-value maps over a rich set of primitive and structured types and organized in collections. Each document is identified by a unique name in its collection; collections are either top-level collections or hierarchically-nested under a document. We show documents using a json-like syntax, and identify documents by a path of collection and document identifier pairs. For instance, `/rooms/C` identifies a document named `C` in the `/rooms` collection and `/rooms/C/messages/22` identifies a document named `22` in the `/rooms/C/messages` collection.

In our example chat room application:

- Each user has a unique id, such as `owl`, and is represented by an account document `/users/owl`. Each document contains information about the user such as name, email address and location.
- A chat room can be created by any user and has a unique name, such as `C`, and is represented by a chat room document `/rooms/C`, which contains fields such as a description and a reference to a thumbnail image.
- Each message posted in room `C` has a unique id, such as `22`, and is represented by document `/rooms/C/messages/22`. A message is a text string that can mention (and automatically notify) other users, even if they are not currently a member of the chat room. For instance, a message whose document is `{ sender: "fox", text: "I saw @owl yesterday", mentions: ["owl"], time: 2026-01-01 23:22:11Z }` and which is sent to chat room `C` will notify all members of the chat room, and also `owl`.

A Firestore query can run standalone, to return the result from a strongly-consistent snapshot of the database, or as a realtime query. The latter returns an initial result from a strongly-consistent snapshot, and then streams a sequence of updates (additions, removals, changes) as the Firestore database gets modified. A client can issue multiple realtime queries, which are grouped on one connection. Updates are delivered with guarantees giving the user a strongly-consistent snapshot of the results across all those queries.

For the chat room application, the main realtime queries, presented with a sql-like syntax, are:

- `select * from /rooms order by name` – observe set of chat rooms.
- `select * from /rooms/C/messages where time > T order by time desc` – listen to recent messages in room C; we assume that the user has joined chat room C.
- `select * from /**/messages where id in mentions and time > T order by time desc` – listen to recent mentions of my id. The syntax `/**/` indicates a collection group query, allowing one query to read from all collections of the same name.

Using the Firestore SDK, application code can easily set up callbacks that trigger when the watched data changes, which in turn can drive UI rendering or other application logic. This simple example in Javascript shows a realtime query connected to a callback that logs every time the data changes, using the first query from the list above:

```
db.collection("rooms").orderBy("name")
  .onSnapshot((querySnapshot) => {
      querySnapshot.forEach((doc) => {
        console.log(doc.id, " => ", doc.data());
      });
  })
```

Database updates to the chat room data may change the results to the above queries through the creation of a new room, the addition of a message to a room, and the deletion of a room or a

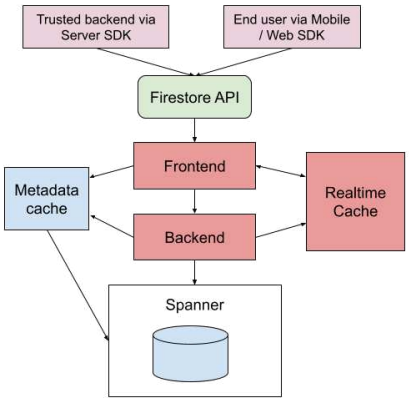

Figure 1: Firestore architecture

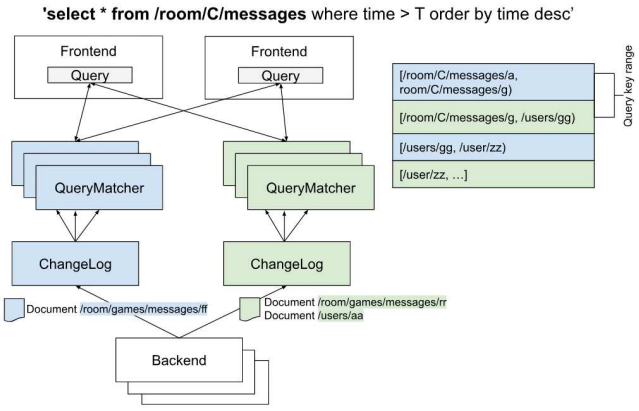

Figure 2: The Real-time Stack with ChangeLog and QueryMatcher tasks organized into groups, with example query and sharding.

message. Strong consistency across multiple realtime queries that are grouped together is critical for accurate behavior and smooth user experience. One example of inconsistency would be if the user adds themselves to chat room `C` and the messages from `C` are now shown in the result set of the second query but `C` is not in the result set of the first query.

Firestore's realtime queries support field comparisons with a constant (all but one must be equalities or `in`), conjunctions and disjunctions, orders, limits and offsets. Joins, aggregations, negation or arithmetic operators are not supported. This limited functionality is selected to support low-latency delivery of updates to queries after database changes. Firestore query updates can nearly always be computed directly from the database's change stream. This efficient execution also makes Firestore's pay-as-you-go query pricing model possible: users are charged a fixed price (a few cents / 100k) for each query result and update. While this feature set is well-suited for OLTP-based application development, it is not for use cases like complex analytics. Developers often adapt their schema design to be able to use realtime queries efficiently, for example by denormalizing their data.

3 ARCHITECTURE

Firestore's architecture has been detailed in our earlier paper [22]. In Figure 1, each of the four rectangles — Frontend, Backend, Metadata Cache, and Realtime Cache — comprises up to thousands of tasks created in each region served by Firestore. The arrows here depict the request (not response) flow. Realtime queries from end-user devices anywhere in the world can access any given Firestore database that is located in a Google Cloud geographical region or multi-region. All Firestore databases are housed in a few underlying Spanner databases, and that allows Firestore to benefit from Spanner's scalability, transactional capabilities, and strong consistency guarantees.

The Firestore service leverages Google Cloud networking infrastructure to route requests to tasks (Frontend/Backend/Spanner) in the "home" region where the database is located. As explained in [22] the Firestore stack is multi-tenant. This means a given task can process traffic to any database that is located in that region, which is key to Firestore's serverless scalability. Tasks are created and shutdown by Google Cloud's auto-scaling infrastructure [30].

The Real-time Cache from Figure 1 serves all realtime queries. It interacts closely with the client SDKs and has three major components, shown in Figure 2 – the Frontend, the QueryMatcher and the ChangeLog. We briefly describe them, and how their interaction with other components enables Firestore's realtime queries.

- End-user devices run the applications built using **Firestore Client SDK libraries**, which are available for multiple device environments (iOS, Android, web, etc). The library maintains a long-lived bidirectional connection to a Frontend task, and uses it to issue realtime queries and receive realtime notifications.
- The **Frontend** serves as an API endpoint for applications to connect to and manages the query lifecycle for the connected devices. When a new query arrives at a Frontend task, it first relays it to the Firestore Backend to retrieve an initial result set, and then establishes connections to one or more QueryMatcher tasks to receive any relevant updates that change the initial result set. The Frontend is responsible for handling limits, sorts and offsets for realtime queries which require visibility into the full query result set.
- The **QueryMatcher** receives a stream of changes based on the modifications transpiring on documents in the Firestore database, matches them to the filters of any live realtime queries using the ChangeStream Filter. The stream of changes, aka change stream, is sourced from the ChangeLog.
- The **ChangeLog** converts all executed database write operations into an outgoing change stream. Unlike an internal commit log, this change stream is not directly integrated with the underlying Spanner database. Instead, for each Firestore write, the Backend employs a two phase protocol to communicate the attempt and outcome of the write with the ChangeLog, ensuring consistency. The direct integration into the stream of commits is what fundamentally allows low latency updates to clients as data changes.

The information flow from the ChangeLog via the QueryMatcher to the Frontend includes the before and after state of each modified document, together with time advancement information that is necessary to ensure consistent ordering of changes. The highest and most recent time (commit version) for which all changes have been received is referred to as "*safetime*" hereafter.

To provide scalability, the system is organized in a tree-like fashion. QueryMatcher and ChangeLog tasks are organized into groups with one ChangeLog feeding multiple QueryMatcher tasks. The overall keyspace of the documents in a Firestore database is automatically sharded based on load, with multiple shards assigned to one group (a ChangeLog task and its QueryMatchers), allowing write operations to scale horizontally. Read (query) scalability is achieved by load-balancing queries across Frontends, and further balancing connections from the Frontends across the QueryMatcher tasks serving the same shard. Figure 2 shows two such groups colored green and blue, and the mapping of keyspace shards of a single database mapped to the groups. As explained later in Section 7.3, Firestore tasks are multi-tenant, and that means keyspace shards from multiple databases can get assigned to the same group.

We look at these components in more detail in the next sections.

4 THE QUERY LIFE CYCLE

In this section, we detail the life cycle of a realtime query, from its initiation to how its result set is subsequently kept up to date by realtime notifications delivered in a consistent fashion. This is accomplished by the flow of information in both directions across the entire stack: the initialization of various data structures in each component, the processing of writes to the database, the matching of those writes with the realtime query, and the propagation of commit timestamps to achieve strong consistency.

4.1 Firestore Client SDK

As mentioned earlier, the end-user devices run applications that are built using the Firestore client SDK library, providing functions allowing them to interact with data stored in Firestore. The SDK maintains an on-device document cache and establishes a long-lived HTTP connection to the Firestore API service, over which it manages one or more realtime queries from the application to the Firestore database. The SDK combines information from its cache and responses from Frontends to provide the best of both worlds: quasi-immediate results, and low-latency strongly consistent results across all the requested realtime queries (when not offline).

When a realtime query is initialized, the SDK computes and returns a result set from the on-device document cache even before it sends the query to the Firestore API. That initial result set gets updated whenever results are received subsequently from the API service. If the user application issues a write, the SDK first updates its on-device cache and the results sets of any affected realtime queries before it asynchronously transmits the write to the API service. The rest of this section describes how the API service manages the life cycle of a realtime query. After which, Section 6 presents the inner workings of the SDK library and in particular how the caching approach allows applications to operate in offline mode.

4.2 Frontend

When the Frontend receives a new realtime query from the client SDK, it first retrieves a base set of matching documents by sending the query to a Firestore Backend task. During this time, the query is in *polling mode*, and we call it a polling query. Once polling completes and an initial result set has been delivered to the client, the Frontend transitions the query to *streaming mode*. To accomplish that, it first computes the query's keyspace range, which can be a single document, a collection, or a collection group. It then refers to a metadata service to find the set of shards covering that keyspace range and the QueryMatcher tasks that currently own those shards, which may change at any time given the dynamic sharding. The Frontend forwards the query to a QueryMatcher task, which in turn is responsible for matching the query with upstream document changes it receives from the ChangeLog in real time. We detail this matching process in Section 4.3.

For each streaming query, document changes are sent to the client in commit order. However, QueryMatcher tasks process document changes in arrival order rather than commit order, and each QueryMatcher task may progress at different rates, so it is the Frontend's responsibility to create commit-ordered output from multiple out-of-order inputs. To accomplish this, the Frontend keeps a commit-ordered buffer of the document changes it has received. Alongside the document changes the stream protocol also provides time advancement information (safetime) indicating that all document changes within a shard range and up to a commit version have been received. As the safetime's minimum common commit version across all shards of a query increases beyond a buffered document change's commit version, the Frontend is guaranteed that all relevant document changes have been received, so it removes the change from the buffer and forwards it to the client. In order to provide a consistent view across all queries in a connection, the Frontend also sends *global snapshot* messages to the client indicating that all active queries on the client connection have progressed to the snapshot's commit version. The challenge and solution for providing snapshots is the same as for single query commit ordering.

Queries using limit and offset require additional coordination. For these queries, the Frontend maintains metadata tracking the size, keys, and boundaries of the result set; when changes from the QueryMatcher cause a document to leave a full limit result set the Frontend issues a polling query to fill in the gap.

The change stream protocol also allows the QueryMatcher to detect when the stream of changes from the ChangeLog for one or more shards is incomplete for some time range, due to task failures or other causes (more details below). In those scenarios, the QueryMatcher rejects any affected queries, and the Frontend keeps the queries in polling mode until it observes read times past the end of the time ranges of the incomplete shards. This mechanism enables the system to continue to serve consistent results transparently to the end-user.

4.3 QueryMatcher

In the QueryMatcher it is critical that given a document change, that consists of the pre- and post-mutation images of the document, the system can efficiently find all matching queries to notify. A

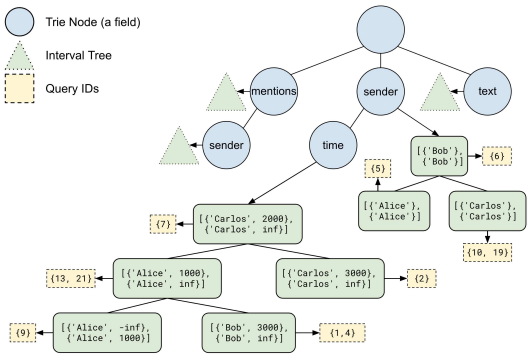

Figure 3: ChangeStream Filter

brute force approach is to iterate over all queries, and for each query check the document change against the query's filter to see if it matches. This is obviously inefficient since the processing cost of each document change by a QueryMatcher task is directly proportional to the number of active realtime queries that are connected to that task.

The ChangeStream Filter shown in Figure 3 is an in-memory data structure that reduces this cost, to be proportional to O(matches) + O(log queries), assuming that the queries for a specific collection usually share similar combination of fields, but with differing filter values. It is organized as a map from lists of fields (from the predicates of the realtime queries serviced by this QueryMatcher task) to interval trees:

```
{ [“mentions”] : <interval_tree1>,
  [“mentions”, “sender”] : <interval_tree2>,
  [“sender”] : <interval_tree3>,
  [“sender”, “time”] : <interval_tree4>,
  [“text”] : <interval_tree5>,
  ... }
```

Each *interval tree* in the map identifies queries whose filter is true if the tuple of corresponding field values is within some interval in the tree. The intervals are generated from the filter conditions of the queries, for instance the filter `where sender = "Carlos" and time > 2000` is represented by the interval $(("Carlos", 2000) - ("Carlos", \infty))$ under map entry `["sender", "time"]`. This representation is sufficient to exactly represent Firestore's filters that allow conjunctions and disjunctions of comparisons of a field against a constant, where the comparisons are equalities or `in` and at most one relational operator.

We find the set of matching queries for a document by looking up tuples of its field values within each entry of the map. The difference between these sets when applied to the pre- and post-mutation image of a document then efficiently gives us the queries currently matching or formerly but no longer matching the document.

For traversal efficiency, this map is implemented as a trie; it organizes the map entries by the shared prefixes of the keys. Each node contains a single field. The path from the root to any node creates a list of fields, and represents a key in the index map. The node also points to the interval tree corresponding to that list of fields.

The interval tree is an AVL tree sorted by the minimum value of each interval, and each node is augmented with the maximum value in its subtree to allow for efficient searching. Each node in the interval trees tracks the IDs of all queries that match its interval.

5 LIFE OF A WRITE

Next, we present how writes to an underlying Firestore database are converted into streams of changes that feed into the QueryMatcher. This requires a serialized log of all mutations to a Firestore database, which is currently not provided by Spanner. In theory, such a log could be designed as a new Spanner table [12] but the realtime stack's needs would make it prone to hotspotting and high costs. Instead, the ChangeLog component collaborates with the Backend to efficiently provide this functionality while avoiding such problems.

The ChangeLog ensures that as mutations are committed to Spanner, the upstream components are provided with a complete, consistent, and ordered sequence of the changes, or notified when something has happened that prevents a consistent view. There is no physically persisted changelog – the combination of changes and time messages allows the upstream components to rebuild the relevant log of changes with strong consistency guarantees. To accomplish this, we use a two-phase Prepare/Accept protocol, where the ChangeLog tracks any open transaction for the shard ranges it is responsible for.

The ChangeLog continuously publishes "heartbeat" messages upstream at a regular cadence to denote the progression of safetime for all the shards it owns, except when overridden by in-flight transactions. A write request to a Firestore database results in mutations to one or more documents and their associated index entries; it is executed by a Backend task as a read-write transaction to the underlying Spanner database. As shown in Fig 4, before the commit for the transaction is issued to Spanner, (1) the Backend sends a Prepare request to the ChangeLog containing the document key(s) in the transaction. (2) The ChangeLog determines the lower and upper bounds for acceptable commit time based on a TrueTime [13][1] call and responds to the Backend. These time bounds are critical for safetime advancement and error recovery. (3) At the same time, the ChangeLog publishes a "blocked shard" message upstream, indicating the inflight transaction and that safetime advancement for the shard range(s) involved in the transaction should be held up at the lower bound, overriding the heartbeat timestamp. (4) The Backend issues the commits for the transaction using an internal Spanner API that guarantees enforcement of the time bounds. (5) Spanner responds with success or failure. (6) In either case, the Backend sends the Accept message with the document snapshots (before and after) of the mutations and actual commit time to the ChangeLog. (7) The ChangeLog then forwards the committed data upstream (8) followed by an “unblocked shard” message indicating that the transaction has been processed (successfully or otherwise) and undoes the safetime override. In practice, all of the logical messages in this protocol are batched in network-optimized sizes for efficiency.

[1]Although TrueTime was originally invented as a highly available, distributed clock for Spanner, it is now available as an internal API that exposes clock uncertainty, and can be invoked by any service.

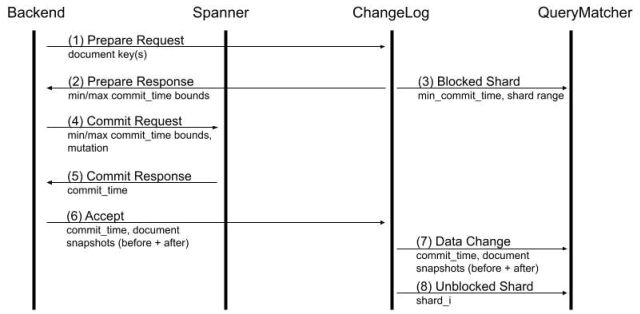

Figure 4: Successful two-phase interaction

An open transaction tracked by the ChangeLog may time out while waiting for the Accept message. When that happens, the ChangeLog is now incomplete for the time period determined by the lower and upper time bounds of the timed-out transaction. This is communicated upstream and the parts of the system managing the queries take appropriate actions to recover. One such example is when a Backend task crashes before it is able to send the Accept message. When a ChangeLog task acquires ownership of a new shard (caused by task failover or re-sharding) it similarly signals that the ChangeLog is incomplete for that key range.

Note that our Prepare-Accept system is different from a traditional 2PC protocol in that the Backend is free to commit the transaction to Spanner as soon as it receives a response to the "Prepare" message. The Spanner transaction is not rolled back even if ChangeLog fails between "Prepare" and "Accept". Instead, the realtime stack detects that it has an incomplete stream of messages, and recovers by failing any affected realtime queries back into polling mode as described above.

It should be noted that when the documents in a Firestore mutation belong in multiple shards, the Backend task executing the transaction task interacts in parallel with multiple ChangeLog tasks owning the corresponding shards. It commits the transaction to Spanner only after it receives successful replies to all its Prepares.

Figure 5 illustrates how data and safetime flows through the system, using the chatroom example. In this example, messages are being written to a collection currently split into two shards. A single query is created to read from the collection, followed by three write operations writing to different shards at different timestamps. The example shows how safetime is propagated through the stack via heartbeats, block and unblock messages, and at what time pieces of data can be released to the client to guarantee a consistent view. Note that we simplify the annotation of timestamps by assuming a discrete set of points in time (for example, t9 is the closest point in time before t10).

6 PROTOCOL CONSISTENCY AND THE SDK

The protocol between a Firestore Frontend and a client (Section 4.2) is a sequence of global snapshot messages indicating that all queries on the connection are up to date at a specific commit version, interleaved with a description of the changes between pairs of snapshots sufficient for the SDK to accurately report query updates. The messages between global snapshots at time T_1 and T_2 are:

(1) A change notification of at least the latest state of each document newly in a query's result at T_2 or changed since T_1. If there are multiple notifications for the same document, they are in commit order.
(2) For each document that is in a query Q's result set at T_1 and not at T_2, either: a notification of its removal, or (in polling mode) additional information on Q's result set that the SDK uses to either (with high likelihood) identify these documents or to invalidate Q. The additional information is, at T_2, Q's overall result count and a bloom filter of result document keys. The bloom filter allows the SDK to identify, based on its known result set from T_1, a count d of documents definitely no longer in the result set: if the count of additional results from change notifications minus d is equal to the change in result count, then the deletes are known exactly. Otherwise Q is invalidated and rerun, transparently to the user.

The protocol does not guarantee that notifications are sent according to an ORDER BY clause or that every query change is sent as a notification, as the first is infeasible during streaming and the second is infeasible during polling. However, the global snapshot, client-side sorting and polling-query validation enable the client to present consistent views of the database state ordered according to the user's expectations.

6.1 SDK Caching and Speculation

A key goal of the SDK, as briefly discussed in Section 4.1, is to provide transparent offline access that works even if the user device is not connected to a network. For example, in our chat application you could imagine a user wanting to read their chat history, or compose new chat messages while being offline and to have these messages be delivered as soon as their device regains network connectivity. To a developer the SDK provides an abstraction where online and offline mode can be treated the same, although with reduced capabilities when the database cannot be reached.

To support this goal, the SDK combines results returned from Firestore with immediate *speculative* results from an on-device document cache. Updates provided by the SDK include their source, so applications can choose, e.g., to ignore speculative results and show only the consistent results provided by the protocol.

The on-device document cache is locally persisted or in-memory (developer's choice) using a light-weight client-side database, such as LevelDB [20] (on iOS), SQLite [19, 21] (on Android) and IndexedDB [1] (in browsers). It caches documents returned from interactions with the Firestore database, mutations issued locally by the user, local indexes to allow faster queries, and auxiliary data for performance optimization and cache management. To provide a consistent view of the data and know the source of results, the cache has two zones: (1) documents received from Firestore until the last global snapshot and (2) pending mutations that have not been processed by the Firestore API service. Notifications from the connection are used to update zone 1 at each global snapshot.

When an application first issues a query it executes immediately against the on-device cache and the speculative results are returned to the application. These results are a combination of known-consistent (though partial) data and the user's recent writes.

Step	Action	Message from Changelog	Frontend buffer	Safetin…
0	The collection **/room/firestore_community/messages** is currently served by two shards (**shard 1** and **shard 2**). Heartbeat process runs at **t0**.	Heartbeat shard 1 (t0) Heartbeat shard 2 (t0)	-	Query safetime t0
1	A new query arrives at the Firestore Frontend -- **select * from /room/firestore_community/messages order by time desc**	-	-	-
2	Polling query runs at **t0** to retrieve the initial dataset for this query.	-	-	Query safetime t0
3	The Frontend splits the query against two different QueryMatchers.	-	-	Query safetime t0
4	Begin a new write transaction, **txn1** Write to **document A** on **shard 1**. min_commit time is set to **t1**.	Blocked shard 1 (t1)	-	Query safetime t0, Blocked shard 1 (t1)
5	Begin write transaction **txn2** Write to **document B** on **shard 1**. min_commit_time is set to **t10**.	No new message, shard 1 is already blocked at lower time.	-	Query safetime t0, Blocked shard 1 (t1)
6	Begin write transaction **txn3**, writing to **document C** on **shard 2**. The min_commit time is set to **t20**.	Blocked shard 2 (t20)	-	Query safetime t0, Blocked shard 1 (t1) Blocked shard 2 (t20)
7	Heartbeat process runs. No new transactions are observed by **t25** on **shard 1** and **t35** on **shard 2**.	Heartbeat shard 1 (t25) Heartbeat shard 2 (t35)	-	Query safetime t0, Min heartbeat safetime: t25 Blocked shard 1 (t1) Blocked shard 2 (t20)
8	**txn1** is committed at **t5**	Document A (t5) Block shard 1 (t10)	A (t5)	Query safetime t9, Blocked shard 1 (t10), Blocked shard 2 (t20)
9	**txn2** is committed at **t30**	Document B Unblock shard 1 (commit time t30)	B (t30) Released: A (t5)	Query safetime t19, Min heartbeat safetime: t25 Blocked shard 2 (t20)
10	**txn3** is committed at **t20**	Document C (t20) Unblock shard 2 at t20 message	B (t30) Released: C (t20)	Query safetime t25, Overall min safetime: t25
11	Heartbeat process runs. No new transactions are observed by **t50** on **shard 1**.	Heartbeat message	Released B (t30)	Query safetime t35 Min heartbeat safetime: t35

Figure 5: Safetime flow example

This fast path ensures a good end user experience when opening an application or making a new request. After the initial data is returned to the application, the query is established with Firestore and the SDK receives the latest version of the documents. On the next global snapshot, the SDK updates the cache and notifies all active queries of their updated, non-speculative result sets.

Similarly, a mutation issued by the application is added to zone 2 of the cache and its effects are speculatively reflected in all active queries before the mutation is sent to the service; this provides immediate feedback to the user. These mutations are sent in issuing order, with subsequent mutations held until previous mutations are either accepted or rejected. If the Firestore API fails a mutation (due to for example deleted documents, changes in permissions etc), the mutation is dropped and the on-device cache and active queries are updated as needed. Applications can register an asynchronous callback if they need to know the outcome of a particular mutation.

If a new query is issued in offline mode, it is treated as best-effort (the only results are speculative). If the on-device cache does not contain a copy of the relevant data, the query will return an empty result. When the device comes back online, the query will be updated with non-speculative results. This heuristic has proved a good fit for most applications.

7 SCALABILITY AND AVAILABILITY

7.1 Automatic scalability

This section explains how the realtime stack scales automatically. It reacts to changes in workload by scaling up or down the number of tasks in its components; per-task resources (CPU and RAM) are fixed empirically. This ensures seamless handling of the organic growth over time of any given application's traffic, peak and trough periods for applications with cyclical traffic patterns, and major business-related events that cause short-term increases in traffic.

It should be noted that Spanner persists Firestore documents (and index entries) as tablets, which are themselves automatically split and merged based on the per-tablet read and write load. In a somewhat similar fashion, but independently and based only on the write load, the write traffic to a Firestore database gets split and merged into shards using the key space of its documents. For example, as shown in Figure 2, with increasing write traffic to the `/rooms/C/messages` collection its key range may split into [`/rooms/C/messages/a` .. `g`), [`/rooms/C/messages/g` .. `z`] across multiple groups to handle the increase in load. Conversely, when traffic to the collection decreases, the shards may merge again to improve efficiency. This process is coordinated using the Slicer auto-sharding framework [3], where each ChangeLog task periodically reports load metrics (traffic rate, payload size) to Slicer, which then decides on split or merge operations. These adjustments are spaced to occur in a few minutes to prevent the system from overreacting to short bursts in traffic since resharding has some overhead.

Realtime query traffic on the other hand, can scale in three orthogonal dimensions:

Direct connections to end-user devices: CPU cycles for Firestore frontends are primarily consumed by active connections, proportional to the number of documents returned by polling queries and incremental updates. The Frontend component scales horizontally based on CPU and memory usage enabled by Google Cloud's auto-scaling infrastructure [30].

Polling requests due to new queries or re-polls: The volume of polling queries is determined by end user activity, starting or using the application on their device. A polling query is the same as a non-realtime query and depends on the scaling capabilities of the Firestore Backend component, along with Spanner's built-in load balancing and splitting.

Fanout to query matching: The task fanout from ChangeLog to QueryMatcher to Frontend enables notifications to scale effectively to numerous end-user devices. The mapping between ChangeLog and QueryMatcher tasks within a group is currently set statically to 1:3 driven by current production demand. Section 9 discusses

mechanisms we use to handle unusual workloads, which allow for custom QueryMatcher fanout configurations.

7.2 High Availability

The main challenges to availability come from problems typical to distributed cloud systems – node or cloud zone failures, task failures or unexpected restarts, networking problems, etc. To an end user, availability problems in Firestore manifest as either increased latency or failed operations. In our chatroom example, a user may not be able to post new messages or may experience delay when receiving new messages. This section discusses some of the mechanisms used to improve the availability of the realtime stack.

The ChangeLog's availability is critical to a Firestore write since the Prepare phase (steps 1 and 2 in Figure 4) of the two-phase protocol needs to complete before the write can be committed to Spanner. An unhealthy or failing ChangeLog task would delay or impact Firestore write operations to documents that lie in the shards owned by that task.

To protect against this type of failure, ChangeLog tasks are organized into replica groups distributed across zones according to the region or multi-region's topology. Shards are assigned at the replica group level rather than to a single ChangeLog task. Within each replica group, the ChangeLog tasks use Google's Chubby-based leader election service [8] to elect one task that handles all incoming writes for the group's assigned shards. If the leader task fails, restarts, or is deemed unhealthy a new leader is elected. The non-leaders in a replica group do not take part in the transactions and have no notion of current change log state. These elections typically complete within 100ms (p50) in single-region deployment (p99 is 1.5s), and 400ms (p50) in multi-regions (p99 is 2s), which often is quick enough that a Backend task can resume an in-flight Firestore write operation by retrying the two-phase protocol with the newly elected leader and completing the write with minor latency impact.

To achieve sufficient availability of ChangeLog leader tasks the system uses a combination of a highly available leader election service and effective task protection and recovery techniques. Leader election is provided by a quorum of 3-5 Chubby cells, each cell itself a zonal service composed of multiple Chubby servers, distributed according to the topology of the Firestore region or multi-region. The resulting availability of the leader election service is significantly higher than Firestore's own SLA, leaving ChangeLog task failure as the main availability concern. We address this concern with flow control and load shedding as well as a collection of heuristics allowing the leader to release leadership or self-terminate under certain conditions such as stuck threads or failure to reach critical dependencies like Slicer or Spanner. In addition, active probers of every ChangeLog leader provide fast detection of an unresponsive ChangeLog task, and are also able to trigger controlled actions to force an unhealthy task to restart.

The deployment of both Changelogs and Chubby in the zones that make up a cloud region is explicitly aligned with Firestore's Spanner topology, and the election of Changelog leaders follows the similar leader election that Spanner uses as much as possible. This is done mainly to optimize write latency and is particularly critical in multi-region deployments, but the failure domain alignment also means that even though successfully reaching the Changelog is an added requirement for every write operation, in the case of problems affecting critical infrastructure in entire zones or regions it typically does not introduce new failures scenarios that the existing dependencies did not already face.

The availability of the ChangeLog and the QueryMatcher tasks also affects the state of the realtime queries. As described earlier, task failures can cause realtime queries to fall into polling mode, followed by replay catch-up before they can transition into streaming mode. Although mostly transparent to the end users, this may manifest as a temporary increase in application latency. The push-back and flow control mechanisms implemented in the polling path have been successful at mitigating the risk of resource exhaustion coming from sudden polling spikes (see evaluation section).

In the case of a Frontend task restart or failure, the SDK on the user device reconnects and resumes the query transparently. This is no different from network connectivity issues between the Google Cloud network end-point and user device (say, due to the user device entering and exiting a tunnel). As described in Section 6, the SDK's on-device cache provides a seamless experience to the user even while the device is offline. The only effect on the end user is increased latency for reads that cannot be serviced by the on-device Client SDK cache.

Tasks get restarted for other reasons too, such as during a release rollout of a new version of a Firestore component or during the spin-up of a new task when a component scales out. Also, Slicer may decide to split, merge, and change assignment of shards to tasks. In all such cases, the system is designed to minimize the impact to realtime queries. For example, when a shard is split the ChangeLog signals the upstream QueryMatcher tasks, which in turn informs the Frontend tasks with affected realtime queries that map to the shards being reassigned. The Frontend tasks transition the affected queries to polling mode and then re-route the queries to the QueryMatcher tasks that now own the re-assigned shards.

7.3 Multi-tenancy and performance isolation

The realtime stack is fully multi-tenant like the rest of Firestore, which means that each task handles traffic to/from many different databases owned by different customers in that region. Although it yields major efficiencies in resource utilization, multi-tenancy brings one important challenge: heavy traffic to one database may degrade the performance (higher latency and/or error rates) of the traffic to another database.

The general approach to provide performance isolation between databases is to apply loose resource constraints for traffic on a per-database level in every task to ensure fairness. The size and number of tasks in each component are provisioned with some extra headroom, which allows Google's auto-scaling framework [30] sufficient time to react to traffic changes to avoid impacting the performance across all databases. Both Autopilot and Slicer [3] are designed to react with some time lag (to avoid overreaction), which means that it is possible for some incoming requests to get throttled during those time lags if a given task hits its resource limits (CPU, memory, bandwidth etc). As described in our earlier paper [22], Firestore requires conforming traffic to grow progressively: increase at most 50% every 5 minutes, starting from 500 QPS [11]. The thresholds for per-database constraints that could result in load

Table 1: Notification latency breakdown.

	p50	p95
Commit to message accepted by Changelog	3 ms	30 ms
Commit to message reaches QueryMatcher	10 ms	52 ms
ChangeStream Filter execution	12 us	75 us
Commit to message reaches Frontend	21 ms	100 ms
Commit to Snapshot sent	15 ms	125 ms

throttling or shedding are implemented appropriately–aggressive thresholds hurt customers and may even confuse the input signal to Autopilot or Slicer whereas lenient thresholds may weaken the isolation protection. The thresholds are determined via experiments with both test and production traffic. We discuss this further in Section 9.

8 EVALUATION

In this section, we evaluate the key aspects of the performance of Firestore's realtime queries: its scalability, latency, isolation ability, and failure recovery. We first provide some production data, followed by two experiments with synthetic benchmarks against a production Firestore database hosted in the us-east4 region. The load in these benchmarks is larger than more than 99.99% of active production databases, though it's still an order of magnitude smaller than the largest production databases. These experiments were conducted during times of average daily production load in that region. We share metrics explaining the observed performance, and provide detailed analyses to showcase the effectiveness of the previously mentioned architectural choices and explain the factors contributing to its success. Lastly, we analyze the behavior of our production system during maintenance operations in the europe-west3 region, showing how it responds to task failures.

8.1 Production workload characteristics

At the time of this writing, Firestore with its realtime stack is available in 45 cloud regions across the world, with us-central1 (homed in Iowa, United States) being the largest. It serves over 8 million databases and 1.5 billion monthly active end users. Approximately 69% of active Firestore databases use realtime queries, and 21% of query results across all Firestore queries are delivered via the realtime query API; 94% of the realtime notifications are new or updated documents, 6% deleted or no-longer-matching documents.

Fanout from write-to-read has a large variance across databases. On average a write translates to 2.1 realtime messages sent, but databases serve dramatically different traffic shapes. The 10 largest databases by fanout have an average write to read ratio of $1:12,000$, peaking around $1:100,000$.

Table 1 breaks down observed notification latencies as time from a commit completes until the message has reached the various components, as measured in Firestore single-region deployments. The counterintuitive data where commit-to-snapshot is lower than commit-to-frontend is explained by fanout in the frontend, where the former is dominated by high-fanout cases that are typically small low latency notifications. The total latency does not include the time it takes for a message to reach an end user's device outside of the Google network.

8.2 Query Scalability

This experiment evaluates how Firestore scales as the number of realtime queries grows. A load test is configured to mimic the chatroom application with a total of 500 active chatrooms, with an increasing number of participants starting up the application on their end-user devices, while sustaining a steady write workload to each chatroom. As described earlier in Section 2, all messages in a chatroom are organized in one collection, for example chatroom C's messages are in `/room/C/messages`. In each chatroom collection there is also a "metadata" document, storing the last time a new message was added to the chatroom.

We measure and report Firestore performance during a 2000 second time period. Each user's application creates two realtime queries: A query to fetch all recent messages of one chatroom that they follow (`select * from /room/C/messages where time > T order by time desc`) and a query with a `where` clause to fetch all recent chat messages that mentioned them from any chatroom (`select * from /**/messages where I in mentions and time > T order by time desc`).

During this experiment, the benchmark mimics the start up of 50 new instances (using different user ids) of the application per second, i.e., 50 new connections and therefore 100 new realtime queries per second connecting to the Firestore database. Once started, each instance of the application and its realtime queries stay active, resulting in a peak of 100,000 concurrent connections and 200,000 realtime queries by the end of the experiment.

During this time, each chatroom receives two new changes every second, one new message document creation and an update to the metadata document. All users following that chatroom will receive a notification containing the new message document and the latest metadata update (2 messages per second). The change also sets the "mentions" user id field to a random user id so that a specific user will receive a document change notification as well.

The line in Figure 6 (scale on the right) shows that the number of real-queries in streaming mode increases smoothly over time. In other words, all newly created queries switch immediately from polling into streaming mode once, and remain so subsequently. The stacked areas in the same figure (scale on the left) show the total number of messages delivered to all instances of the application over time; broken down by snapshot messages and data messages. Had Firestore not dynamically scaled to the experiment's traffic, we would have observed higher latencies (see next section) caused by throttling or overload in the various layers across the stack.

8.3 Notification Latency

We use the above experiment to study the latency of realtime queries. Figure 7 plots the p50, p90, and p99 notification latencies across all queries. Notification latency is measured from when Spanner acknowledges the successful commit of a mutation that adds new messages to the chatroom `/room/C/messages` (and therefore belong in the result sets of the realtime query from all clients) until when the corresponding notifications have been received by *all connected clients*. All three plots in the graph remain stable even as the number of client connections and realtime queries increases with time, which demonstrates how Firestore scales transparently with increasing client load.

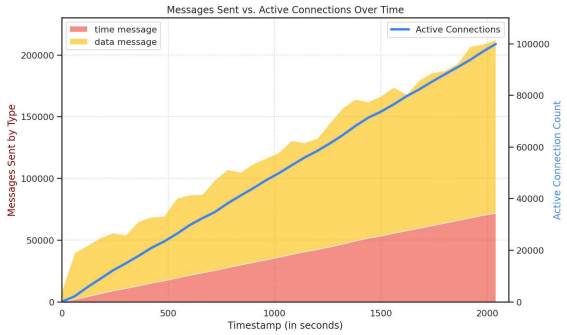

Figure 6: Number of messages delivered and the number of active connections.

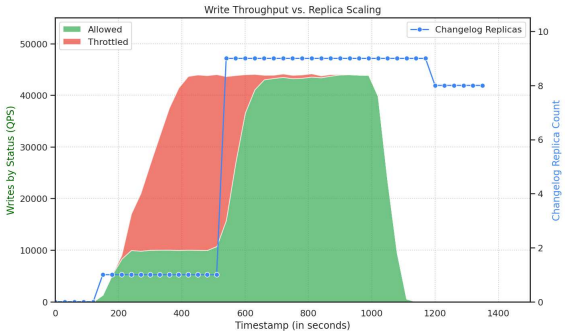

Figure 8: ChangeLog group count vs write throughput for the Bully database.

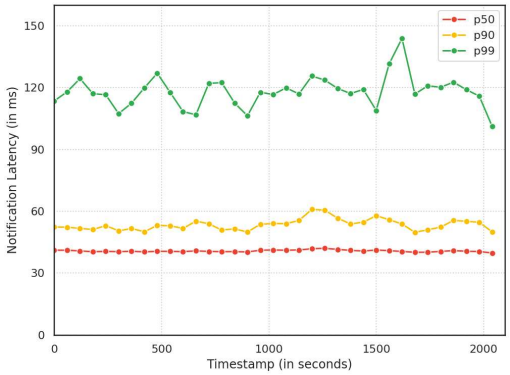

Figure 7: Notification latency

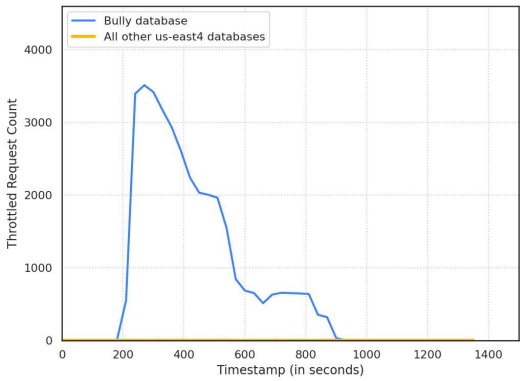

Figure 9: Throttled writes operation to the Bully vs all other databases in that region.

We also observed the latency of queries moving from polling to streaming mode, which mirrors the experience of end-users when they open the application for the first time. This latency is measured from the instant a new client connection is created by the load generator, until both its realtime queries have transitioned to streaming mode and have a consistent up to date view of both queries in the application are available to the client (and its queries have switched to streaming mode). Here we measure p50 latencies ranging from 130ms at the start of the experiment to 260ms at the end. The latency here is dominated by the polling query and it increases with time since the result set for the gets larger as more messages get posted to all chatrooms (by the end of the test each chatroom collection contains 2000 messages and one metadata document), but the time to switch from polling to streaming mode remains unchanged with time and the increase in connections and queries.

8.4 Write Scalability and Performance Isolation

Firestore's architecture is multi-tenant, handling traffic from clients to hundreds of thousands of databases within a single region. In this experiment, we study Firestore's ability to isolate the effects of *non-conforming traffic* to one database, preventing it from impacting the traffic to other databases. Non-conforming traffic violates the "500/50/5" rule [11] described earlier in Section 7.3. We created a Firestore database called the "Bully" in the us-east4 region, to which we applied a non-conforming write load that ramped up from 0 to 45k new document writes/sec, accelerating at 183 $writes/sec^2$, after which the write load runs until the 1000s mark at a stable 45k/s. The Bully project mimics traffic spike to a database that becomes suddenly popular due to some real-world viral event.

The stacked areas in Figure 8 (scale on the left) plot the rate at which write operations to the Bully database were allowed or throttled (by the Backend component), while the dotted line (scale on the right) shows the number of ChangeLog and QueryMatcher groups that are assigned the document key space shards of the Bully database. As explained in Section 7.1, we expect the key space to get split (merged) across groups with increasing (decreasing) load. The resharding adjustment takes ~5 mins to occur, which aligns with the 500/50/5 rule. As the dotted line shows, the Bully database key space is initially handled by one group. It takes ~5 min to adjust to the traffic spike, after which the key space is sharded across 9 groups. During that 5 min time interval, the stacked areas shows how the allowed write load stays flat while the rest of the writes are throttled. Once sufficient group capacity is created, the throttling ceases and all write operations succeed.

Figure 9 shows that the throttling impacts only the Bully database, which is depicted by the blue line. The yellow line (staying at

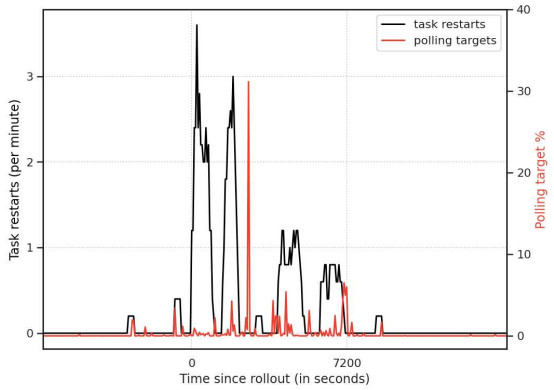

Figure 10: Task restart rate of QueryMatchers and impact on percentage of queries in polling mode.

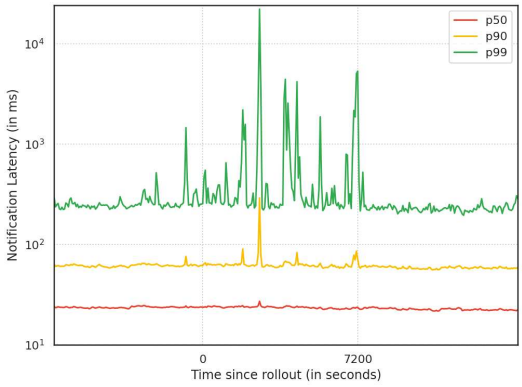

Figure 11: Notification latency (log scale) during a two hour rollout of QueryMatchers.

zero the whole time) shows that operations to all other customer databases in that region were not impacted, thereby showcasing Firestore's ability to isolate the effects of non-conforming traffic. We also observed unchanged steady latency metrics for all other databases during the time of the experiment.

8.5 Availability

To study the impact of task failure, we show data from a periodic rollout of QueryMatcher software: when a QueryMatcher task restarts, all the realtime queries it handles are briefly forced into polling mode. Our software rollout does not instruct tasks to hand off work before shutting down. Thus, this mirrors a number of failure modes, primarily crashing QueryMatcher or crashing ChangeLog tasks, or other issues that trigger storms of polling queries, but a full rollout is significantly worse than random failures, as all the tasks are restarted over a couple hours.

Figure 10 shows the task restart rate and percentage of realtime queries during a rollout that starts at time 0 and ends at time 7200. Restarts peaks at around one every 17s, and the percentage of queries in polling reaches a brief peak of 31%. The impact is acceptable (Figure 11): overall median notification latency is essentially unchanged, and the system always recovers quickly from the p90 and p99 latency excursions as the queries return to streaming mode in less than one minute (our system's monitoring interval).

9 LESSONS IN PRACTICE

Firestore with its Realtime stack is available today in 45 cloud regions across the world, with us-central1 (homed in Iowa in the United States) being the largest.

The automatic scalability mechanisms outlined earlier have guaranteed that customer applications keep working without interruption as traffic grows and changes. We observe different types of growth patterns for customers depending on the type of applications they deploy. Applications see growth in some combination of number of clients, number of realtime queries, and number of write operations, often with peaks and troughs at different times during the day. The Realtime stack being able to scale independently in multiple dimensions has been critical to our success.

A common challenge for supporting large numbers of end-users is spikes in traffic. This can be caused by a viral event – thundering herds of users appearing suddenly due to a once-in-a-year Superbowl commercial or a holiday sale, or more commonly due to application design choices – big batch jobs that run at midnight or a periodic event triggered by an application, such as Be Real [6]. When these events can be predicted, the Firestore SRE team works with the customer to preemptively increase capacity for their database. Otherwise, the Realtime stack's approach to scaling up or down the system in minutes has proven effective in these scenarios. Being entirely and inherently multi-tenant improves the likelihood that a large event for one customer becomes a small blip in the context of the overall shared machine capacity provisioned to that region.

Section 7.3 discusses the downside of multi-tenancy, viz. noisy neighbors. Consistent performance is extremely important for many of our customers, and therefore performance isolation has been addressed as a fundamental design principle.

Beyond the automatic defense mechanisms covered in earlier section, we have also deployed "isolation pools" – essentially parallel deployments of the different components in the stack with dedicated capacity, and a set of tools that lets an SRE quickly and temporarily move repetitively non-conforming traffic out from the shared pool. While isolated, it buys time for the engineering team to better understand the traffic and/or reach out to the customer.

In other words, isolation pools allow us to deal with unexpected traffic patterns, when the resource cost estimates of the load shedding mechanisms do not map well to actual resource cost. For example, an early version of the throttling mechanism used the number of documents per write operation as an estimate for resource requirement. This proved insufficient for traffic with very large documents that needed more resources to parse. Similarly, as Firestore propagates full before and after documents to evaluate query predicates, applications that heavily use realtime queries on massive documents have significant network bandwidth and memory overhead. We actively encourage developers to avoid using realtime queries on such documents, as this anti-pattern leads to degraded notification latency and increased client memory pressure. The automatic throttling mechanisms are triggered multiple

times per day without requiring any human intervention. Their effectiveness has been borne out by the fact that SRE intervention and isolation pools are needed rarely, only around 5 times per year.

One interesting challenge with isolation pools is that strong consistency guarantees must be retained for the queries even while non-conforming traffic is migrated. If traffic is moved from one pool of ChangeLog servers to another, it is critical that all upstream tasks consuming updates also migrate, but no data changes are missed during that switch. To ensure this we have added a special pool change protocol to the shard ownership mechanism; we do not present the protocol here for lack of space.

Firestore employs standard production monitoring best practices using Monarch [2] to be able to detect system anomalies, trigger alerts, and have SRE take action, but the fact that the Realtime stack is different from a common request-response serving system has required us to redefine SLOs to be centered around the lifetime of a query and a client connection than individual requests. For reliable monitoring, we augment measurements of real customer traffic with a suite of probers that generate artificial traffic to simulate both typical and atypical customer applications – it establishes long-lived connections and realtime queries, issues write operations and validates that the expected data arrives at the clients. Doing this at sufficient scale to ensure coverage of the entire Firestore deployment has allowed us to detect subtle problems early.

On a related note, the fact that the system is centered around streaming data processing and the rendezvous between reads and writes led us to use a programming model that's different from the common request-response-centric models. Early on, we found that using shared memory and traditional concurrency primitives, such as locks, resulted in our system becoming bug prone – changes to the code caused problems like mutex contention or dangling pointers frequently. We replaced that architecture with an actor model framework based on request queues. This framework has allowed us to produce more obviously correct code by forcing our developers to be explicit about the desired concurrency behavior. Although this model required more work in some cases to avoid bottlenecks and to unblock throughput, it has been a net positive for development velocity.

10 RELATED WORK

In our original Firestore paper [22] we presented Firestore's overall model and architecture; Davisson et al [16] presented our no-downtime migration of existing customers to Firestore.

There are several other distributed databases providing Firestore-like realtime queries. InvaliDB's [35] implementation is closest to Firestore: it has similar query functionality (filters, boolean operators, sorting) and achieves scalability – published results scale up to tens of thousands of writes and queries per second – in a similar fashion by sharding writes across multiple write replicas, and query scalability by replacing individual writes across query replicas. Unlike Firestore, queries are eventually consistent and the sharding appears to be manual. Several open-source databases – RethinkDB [28, 34], Meteor [32], Parse [27] – support realtime queries, but the implementations rely on sending the entire database's change stream to a single node, which limits write scalability. NiagaraCQ [9] provides continuous queries over external data sources (that change relatively slowly). As in Firestore, NiagaraCQ groups related queries so that they can be efficiently executed together, periodically or when data changes.

Both MongoDB (MongoDB Realm [23], now discontinued) and Couchbase (Couchbase Lite [14]) provide local databases with realtime queries and automatic remote synchronization. These differ from Firestore in not providing strongly consistent snapshots, and requiring local replication of the database. For security, Couchbase and MongoDB provide document-level control of what data is replicated, and thus accessible by end-users.

Firestore's realtime queries, and realtime query execution techniques, are similar to materialized views, continuous stream queries and publish/subscribe systems.

A summary of techniques for incrementally updating materialized views from a transaction's change delta can be found in [10]; recently, DBSP [7] has shown an algorithm to incrementally maintain any SQL or Datalog query. Computing Firestore's realtime query updates require only simple techniques and our implementation focuses mostly on achieving scalability in the face of high write rates and large numbers of realtime queries. Firestore's realtime queries are continuous and long-lived, and thus do not belong to any transaction's scope. As a result, and unlike materialized views, there is no requirement to have consistency with the rest of the database (and other views). Instead, Firestore provides consistent, timestamped snapshots of the state of a set of realtime queries. This frees our implementation to skip some updates, rerun queries from scratch at the current time, etc.

Like Firestore's realtime queries, continuous stream queries [5, 33] continuously report a query's latest state. However, continuous queries usually assume some set of restrictions on queries (e.g., queries over windows of recent data) or the database (e.g., append only) so that queries can always be efficiently executed incrementally (without requiring access to the full data set). Stream processors [4, 18] are similar but replace the query with arbitrary code.

Publish/subscribe systems [17] often support subscribing with query-like filters on published items. AWS AppSync [31], MongoDB [24] and OrientDB [25, 29] support subscriptions with filters to database change streams. Zanzibar [26] also uses Spanner TrueTime guarantees to provide strong consistency guarantees for ACL checks. Additionally, it allows watching (subscribing) to the ACL change stream of a particular namespace in near real-time, based on a sharded, TrueTime-ordered change-log persisted to Spanner.

Cugola and Margara [15] and Wingerath et al [36] provide in-depth surveys of these realtime data processing systems.

11 CONCLUSION

In this paper, we developed on the original Firestore paper [22] to present the internal details of the production-grade Firestore realtime stack that provides strongly consistent realtime queries with low latency, together with automatic scaling, high availability, and performance isolation from noisy neighbors. We explained how the Firestore SDKs on the client provide a smooth application experience to an end-user even across loss of network connectivity. We analyzed the performance of the realtime stack, presented relevant production data and important lessons in practice.

REFERENCES

[1] 2025. *Indexed Database API 3.0, W3C Working Draft.* Retrieved 2026-07-08 from https://www.w3.org/TR/IndexedDB/

[2] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Adam Tart, and Nick Taylor. 2020. Monarch: Google's planet-scale in-memory time series database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3181–3194. doi:10.14778/3181-3194

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 739–753. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[5] Shivnath Babu and Jennifer Widom. 2001. Continuous queries over data streams. *ACM Sigmod Record* 30, 3 (2001), 109–120.

[6] Alexis Barreyat. 2022. *BeReal builds a real and authentic social media platform on Google Cloud.* Retrieved 2026-07-08 from https://cloud.google.com/blog/topics/startups/bereal-creates-reality-based-social-media-using-google-cloud

[7] Mihai Budiu, Leonid Ryzhyk, Gerd Zellweger, Ben Pfaff, Lalith Suresh, Simon Kassing, Abhinav Gyawali, Matei Budiu, Tej Chajed, Frank McSherry, et al. 2025. DBSP: automatic incremental view maintenance for rich query languages. *The VLDB Journal* 34, 4 (2025), 39.

[8] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[9] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 379–390.

[10] Rada Chirkova and Jun Yang. 2012. Materialized views. *Foundations and Trends in Databases* 4, 4 (2012), 295–405.

[11] Google Cloud. [n. d.]. *Firestore in Native Mode: Best Practices.* Retrieved 2026-07-08 from https://docs.cloud.google.com/firestore/native/docs/best-practices#ramping_up_traffic

[12] Google Cloud. [n. d.]. *Spanner: Change Streams Overview.* Retrieved 2026-07-08 from https://docs.cloud.google.com/spanner/docs/change-streams

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, 261–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[14] Couchbase. [n. d.]. *Couchbase Lite: an embedded NoSQL JSON document database designed for mobile and edge devices.* Retrieved 2026-07-08 from https://docs.couchbase.com/couchbase-lite/current/index.html

[15] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.

[16] Ed Davisson, Tilo Dickopp, David Gay, Eric Karasuda, Ram Kesavan, and Vadim Yushprakh. 2024. Transparent migration from Datastore to Firestore. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3960–3972.

[17] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[18] Apache Software Foundation. 2014. *Apache Storm.* Retrieved 2026-07-08 from https://storm.apache.org/

[19] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: past, present, and future. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3535–3547. doi:10.14778/3554821.3554842

[20] Google. 2014. *LevelDB.* Retrieved 2026-07-08 from https://github.com/google/leveldb/blob/main/doc/index.md

[21] D. Richard Hipp. 2000. *SQLite.* Retrieved 2026-07-08 from https://sqlite.org/

[22] 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)*. 3376–3388. doi:10.1109/ICDE55515.2023.00259

[23] MongoDB. [n. d.]. *Realm (Legacy) - Atlas Device SDKs - MongoDB Docs.* Retrieved 2026-07-08 from https://www.mongodb.com/docs/atlas/device-sdks/realm-query-language/

[24] MongoDB. 2018. *Change Streams.* Retrieved 2026-07-08 from https://www.mongodb.com/docs/manual/changeStreams/

[25] OrientDB. [n. d.]. *Live Query - OrientDB.* Retrieved 2026-07-08 from https://orientdb.dev/docs/3.0.x/java/Live-Query.html

[26] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 33–46. https://www.usenix.org/conference/atc19/presentation/pang

[27] Parse. [n. d.]. *Parse Platform.* Retrieved 2026-07-08 from https://parseplatform.org/

[28] RethinkDB. [n. d.]. *RethinkDB: The open-source database for the realtime web.* Retrieved 2026-07-08 from https://rethinkdb.com/

[29] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. 2021. OrientDB: A NoSQL, Open Source MMDMS. In *British International Conference on Databases*. https://api.semanticscholar.org/CorpusID:251648993

[30] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Article 16, 16 pages. doi:10.1145/3342195.3387524

[31] Amazon Web Services. [n. d.]. *Using subscriptions for real-time data applications in AWS AppSync.* Retrieved 2026-07-08 from https://docs.aws.amazon.com/appsync/latest/devguide/aws-appsync-real-time-data.html

[32] Meteor Software. [n. d.]. *Meteor.* Retrieved 2026-07-08 from https://meteor.com/

[33] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. 1992. Continuous queries over append-only databases. *Acm Sigmod Record* 21, 2 (1992), 321–330.

[34] Leif Walsh, Vyacheslav Akhmechet, and Mike Glukhovsky Hexagram. 2009. RethinkDB — Rethinking Database Storage. https://api.semanticscholar.org/CorpusID:58917842

[35] Wolfram Wingerath, Felix Gessert, and Norbert Ritter. 2020. InvaliDB: scalable push-based real-time queries on top of pull-based databases (extended). *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3032–3045. doi:10.14778/3415478.3415532

[36] Wolfram Wingerath, Norbert Ritter, and Felix Gessert. 2019. *Real-Time & Stream Data Management: Push-Based Data in Research & Practice.* Springer.