11-28-2023

Energy Auction with Non-Relational Persistence

Michael Ramez Howard
Portland State University

Energy Auction with Non-Relational Persistence

by

Michael Ramez Howard

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Wu-chang Feng, Chair
R. Bruce Irvin
David Maier

Portland State University
2023

Abstract

As the current landscape for electric vehicles changes, options for remote charging are expanding to keep up. In the United States alone, sales of electric vehicles grew 85% from 2020 until hitting 450,000 units by the end of 2021. While these growing sales are encouraging, commercial charging stations have a long way to go before they are as ubiquitous as gasoline stations are today. The peer-to-peer energy auction helps fill the gap in underserved areas by allowing private homeowners to share their charging facilities with other electric vehicle drivers. The auction framework wraps existing charging outlets with a Cloud-connected microcontroller. These Edge devices communicate with a Cloud message broker for both reporting and session control purposes. Users, both buyer and seller, may interact with the framework through a web-based user interface. The design of this framework provides many challenges, including how to handle persistent storage. The technology used for data storage is key in determining the performance of the auction application and the smoothness of the user experience.

The two technologies considered are a SQL, server-based, structured and a NoSQL, serverless, unstructured database. NoSQL has gained momentum in the last 20 years triggered by the needs of Web 2.0 companies requiring user-generated content, ease of use and interoperability revolving around big data and real-time access. The natural division of management and storage layers allows for a robust serverless implementation: the Cloud service listens for requests and processes using shared and abstracted compute resources. Serverless allows a payment model where each transaction, beyond the free tier of 20,000

writes/50,000 reads per day, is billed rather than paying continuously for a deployed compute instance.

In this study, we investigate Google's Firestore and Cloud SQL MySQL solutions. We pit the newer serverless, non-relational, NoSQL, document-model database against the traditional SQL, table-based, relational server. Both solutions are evaluated for query performance, cost, flexibility and scalability. Through benchmarking, analysis and a deep-dive of system architecture, we answer whether Firestore can support the energy auction persistent storage needs despite the superior query capabilities of MySQL's SQL engine.

I would like to dedicate this thesis to my wife Dianna and two daughters, Yasmine and Myriam. They were patient with me and put up with the long hours spent in research. It was never easy to sacrifice spending time with them and their understanding and support allowed me to move forward.

Acknowledgements

I would like to acknowledge and thank Dr. Wu-chang Feng for advising me throughout this thesis and helping make it possible. His courses on Cloud technology and security were extremely informative and helped guide my path throughout my research.

I would also like to thank the contributions of Dr. R. Bruce Irvin and Dr. David Maier. Both of whom served on my advisory committee and provided motivation and assistance to the research topic.

Finally, a huge thanks to my wife Dianna and daughters Yasmine and Myriam. They were patient with me throughout my long hours of research. It was with their support that I was able to reach the end goal.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	**A**tomicity **C**onsistency **I**solation **D**urability
API	**A**pplication **P**rogramming **I**nterface
AWS	**A**mazon **W**eb **S**ervices
CAP	**C**onsistency **A**vailability **P**artition Tolerance
CRUD	**C**reate **R**ead **U**pdate **D**elete
DBMS	**D**ata**B**ase **M**anagement **S**ystem
DHT	**D**istributed **H**ash **T**able
EC2	**E**lastic **C**ompute **C**loud
EV	**E**lectric **V**ehicle
GCE	**G**oogle **C**ompute **E**ngine
GCP	**G**oogle **C**loud **P**latform
GQL	**G**raph **Q**uery **L**anguage
IaaS	**I**nfrastructure **a**s **a** **S**ervice
IaC	**I**nfrastructure **a**s **C**ode
IoT	**I**nternet **o**f **T**hings
ML	**M**achine **L**earning
NN	**N**eural **N**etwork
NoSQL	**No**t Only **S**tructured **Q**uery **L**anguage
PaaS	**P**latform **a**s **a** **S**ervice
RDBMS	**R**elational **D**ata**B**ase **M**anagement **S**ystem
RPC	**R**emote **P**rocedure **C**alls
SDK	**S**oftware **D**evelopment **K**it
SQL	**S**tructured **Q**uery **L**anguage
VM	**V**irtual **M**achine
YCSB	**Y**ahoo! **C**loud **S**erving **B**enchmark

Chapter 1

Introduction

The era of internal combustion vehicles is winding to a close. Electric vehicles (EV) sales are on a steady upward trajectory. EV-Volumes [14] reports that in 2022, sales increased by 55% over the previous year while the entire auto industry declined by -0.5% over that same period. A large barrier to EV ownership is range anxiety. Commercial charging facilities are not yet as ubiquitous as gasoline stations, opening up a market for private individuals to assist by sharing their personal charging facilities. In this thesis, we detail how the Cloud framework for a peer-to-peer energy auction might work to provide users a means to both share and rent their private charging outlets. Components of the framework include the Edge microcontrollers, web server, API, Cloud Functions, Publication-Subscription service and a message broker. We then dig deeply into the database component which the main research question is formed around.

1.1 Motivations

A critical step in the energy auction is to persistently store data objects representing the charger Edge devices. Users, charging session history, billing details, ratings and reviews, amenities and tourist information are additionally

persisted. Workflows for the auction users include registering a new charger device, searching for available devices and managing a charging session as seen in Figure 1.1.

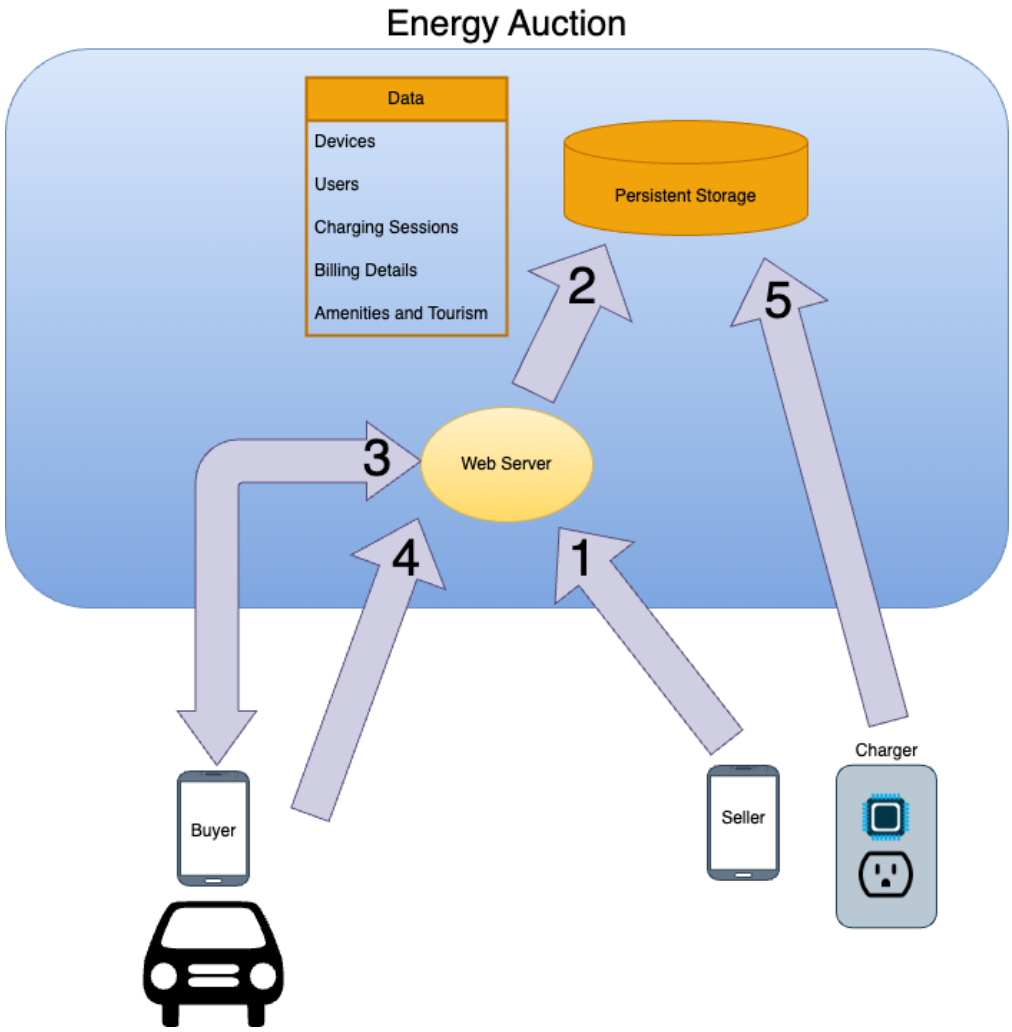

FIGURE 1.1: The energy auction supporting: 1. Device registration, 2. Data storage, 3. Device search and reserve, 4. Session management, and 5. Session details.

Requirements for the persistent data include low read and write latency, the

ability to scale horizontally as data grows, cost, and the ease of evolving the data shape. Administration operations such as create, update and delete operations on a registered device will only be performed by a single user and will not be adversely affected by concurrency locks or extra latency. However, multiple users may query the persistent storage asking for data objects that contain multiple fields within a given range. The read operation latency will be critical in this scenario. The user must be able to request a charger within a given latitude and longitude radius and may need to include maximum distance to a restroom, minimum user rating, type of charger, etc. The query engine must be able to process these patterns of field conditionals. If it can only process a subset, the resulting data reduction must be efficient enough such that the user response is not noticeably delayed.

Another motivation in researching the database technology is flexibility and scalability. The energy auction will grow in iterations, adding features requiring additional persistence. A fixed schema or data shape has to deal with migration when the schema changes. A flexible schema provides the developer with the option to handle the migration in code as the auction framework and feature set evolve. Since the number of users participating in the auction is also evolving, the persistent storage must be able to horizontally scale to accommodate these. For example, an early auction pilot may have 50 users within a single geographical zone. An individual Cloud node with failover and redundancy can safely store all required persistent data and support all incoming query requests within the zone. However, as users spread out geographically, storage nodes will need to scale into these new zones. Also, as data and user request

volume exceed what can be processed by a single node, the storage must partition and scale across a node cluster.

The high-level goals for the persistent storage for the auction are as follows:

1. Low storage cost.
2. Low compute cost.
3. Multi-user query support.
4. Low query latency.
5. Ease of scaling.
6. Ease of schema evolution.

Cost is a huge motivation, driving a design goal of the framework to avoid perpetual, unmanaged compute. The deployment of compute services that are always in operation results in a significant billing increase, since the compute resources are dedicated and always available. Additionally, the type of compute and operating system patches must be managed by the developer. A serverless persistent storage provides a pricing model where the billing only reflects what is used. Thus, queries and storage contribute directly to billing, but idle time does not. However, the serverless model is potentially more expensive, compared to on-demand, if subjected to continuous load.

1.2 Research Question

The aforementioned requirements for persistent storage raise the question of what will work best to support the energy auction. A deep-dive investigation is beneficial prior to developing the auction framework and leads us to

the research question addressed in this thesis: **Will a serverless, NoSQL, non-relational, document-model database (i.e. Google Firestore [8]) support the query, cost and flexibility needs for persistent storage in a peer-to-peer energy auction?** We will endeavor to answer this question through benchmarking latency, estimating costs and analyzing the architecture. To provide a comparison to a well-established solution, benchmarking and analysis is compared side-by-side with MySQL: a relational, SQL, perpetual server provided through Google Cloud SQL [10].

1.3 Context of the Study

Multiple related studies utilized a third-party benchmarking tool such as Yahoo! Cloud Serving Benchmark (YCSB) [32] to establish latency under varying load [22, 2, 13, 25]. In this study, we will focus on a more customized means to measure the database response time.

For both the relational and non-relational databases, we develop client applications that emulate portions of the energy auction data flow. Queries run from these clients perform read and write operations that characterize the latency performance under specific load scenarios. This experimental setup allows the persistent storage component of the auction to be developed and tested in isolation. The solution representing the best trade-offs around cost, latency and flexibility will continue to live on in the auction framework and support the investigation of future research problems.

1.4 Objectives and Contributions

The objectives of this thesis are as follows:

1. Characterize query latency for both relational and non-relational databases.
2. Evaluate cost of both solutions.
3. Explore ease of evolving the stored data structure.
4. Explore ease of horizontal scaling.

Achieving these objectives provides a characterization of the strengths and weaknesses of Firestore's persistent storage solution, both standalone and comparatively against the more established SQL solution. This insight into Firestore's performance and capabilities contributes a known data storage platform to support the future energy auction framework.

1.5 Overview of the Thesis

After initially introducing the topic, the background of electric vehicle (EV) charging and related research is covered in Section 2. Next, we discuss the system architecture of both Firestore and MySQL storage solutions in Section 3. The experiments performed against both solutions are presented in Section 4 along with their corresponding experimental data. The discussion of the experimental results, regression model and future work are detailed in Section 5. Finally, we wrap up the thesis with the conclusion in Section 6.

Chapter 2

Background

2.1 EV Charging

Electric Vehicle (EV) ownership has increased significantly in the United States. The combined sales of EVs and plug-in hybrids nearly doubled from 308,000 in 2020 to 608,000 in 2021 [19]. The EV sales alone grew 85% during that time period while the overall market for light-duty vehicles sales only grew by 3%. Thus, electric vehicles are increasingly taking market share for personal transportation vehicles.

The accelerating EV sales require an acceleration in charging infrastructure deployment. What if, by 2030, a peer-to-peer auctioning system existed such that consumers can turn into producers and sell units of energy? The rising popularity of solar panels [26] allows for the generation of energy as well as reselling energy purchased via a utility grid. Peer-to-peer sales will help meet the increasing demand of EV charging and offers customers alternatives as a part of the evolving sharing economy. Airbnb [1] provides a peer-to-peer service as an alternative to traditional hotels while Uber [29] is a popular alternative to conventional taxi services. The popularity of these services reveals a willingness by the public to use peer-to-peer alternatives and this proposal will focus on the development of a distributed energy auctioning system.

While the majority of charging may be done in the home to facilitate local commutes, a publicly available network of chargers is still necessary for intercity commutes and to accommodate those who do not have charging facilities in their home. As a result of this demand, EV charging companies have rapidly formed and deployed networks of chargers in the United States. Figure 2.1 shows the top companies at the end of 2021 ranked by total number of DC fast chargers.

Rank	Charging Network	DC Fast Chargers (ports)	% of Total
1	Tesla	12,580	58.0%
2	Electrify America	3,112	14.4%
3	EVgo Network	1,711	7.9%
4	ChargePoint Network	1,675	7.7%
5	Non-Networked	909	4.2%
6	Francis Energy	545	2.5%
7	Greenlots (Now Shell Recharge)	477	2.2%
8	EV Connect	183	0.8%
9	EVCS	175	0.8%
10	Blink Network	154	0.7%

FIGURE 2.1: Top 10 fast-charging companies sorted by number of locations [6]. The data was sourced from the US Department of Energy as of Dec 31, 2021 [30].

Since the charging networks as shown in Figure 2.1 are still far from being as ubiquitous as gasoline stations, we propose supplementing them with private property owners who wish to rent their charging facilities through a peer-to-peer energy auction. Figure 2.2 shows the architecture to implement the

auction. A seller will register their 240-volt level-2 charging outlet via an Internet of Things (IoT) microcontroller. The IoT device communicates with a Message Broker within the Cloud framework, which is registered with a dedicated "Publication- Subscription" (PubSub) service. PubSub may send device data through publication requests and return device data through subscriptions. The auction is a simple marketplace that sells power along with a time slot. There is potential to expand to eventually allow customer bidding for these resources.

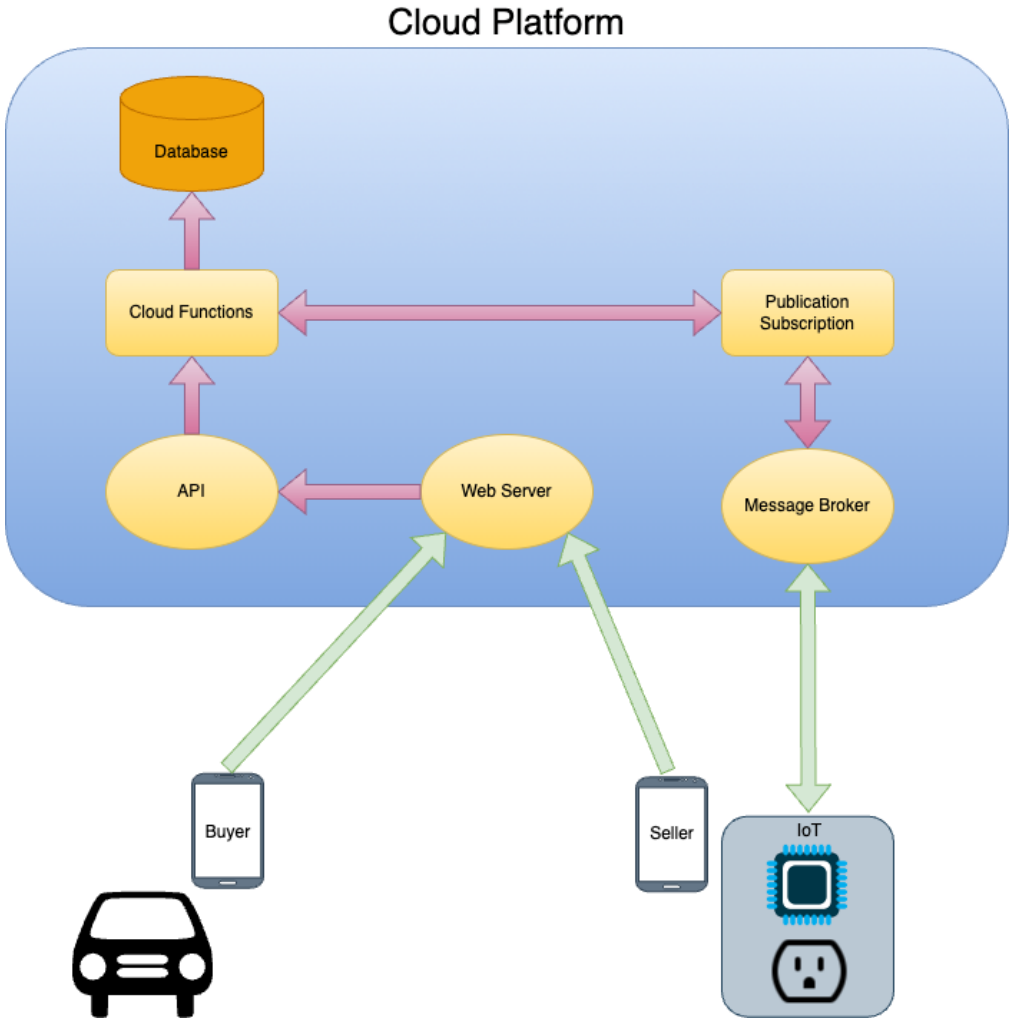

FIGURE 2.2: A Cloud framework to implement the energy auction. A seller registers their IoT device allowing buyers to search and reserve. Cloud Functions publish commands to start and stop the charging sessions on behalf of the buyer. Sellers subscribe to the charging session summary.

Both the buyers and sellers interact with the auction through a web client. Once a seller registers an IoT device, the web API creates a corresponding database object. All requests are processed through Cloud Functions. A buyer will search

for all devices within a given latitude-longitude radius. Within the search results, the buyer chooses the desired device, reserves it (potentially with a bid sequence) and navigates to its location to start the charging session. The Message Broker passes on control messages from the device which are forwarded, via a subscription, to Cloud Functions to modify the device database object. Once the session is completed, a detailed log is sent by the device, which is stored in the database. Additionally, Cloud Functions trigger payment processing via an external payment gateway (e.g. Stripe [27]).

A significant question in this design is what are the persistent storage needs for this auction? Our focus in this thesis is the data flow caused by the aforementioned workflow and performance of the database. We do not know all data storage requirements initially. For example, when this framework is initially rolled out, only charger details such as:

```
name
address
latitude
longitude
type
charging history: date, duration and energy consumed
```

will be persistently stored. As the development progresses, we will potentially add:

```
billing information
user ratings and comments
ammenities
nearby tourism information
user profile
```

Given that the data fields and shape will be evolving; a strict, schematized database will cause additional complexity since existing data must be migrated every time there is a new change, or new tables must be added without duplicating information such as tourism facilities.

Another concern is the scalability. If the auction has 10 sellers and 10 buyers, scaling is not an issue. If this number were to grow to 1,000,000 sellers, the database must be able to horizontally scale device objects across multiple nodes with physical locations across multiple zones and regions. A NoSQL database with a document model allows this scaling with little administration.

Query latency and performance is also a concern. Create, update and delete operations are single participant and can tolerate eventual consistency. However, a read operation needs to query multiple fields, some within a range. If a buyer searches for charger type of level 2 along with a latitude, longitude and user rating range; a SQL query engine can easily handle this and return only matching records. However, a NoSQL database such as Firestore will only be able to query a single field range at a time, requiring additional data reduction at the client or the interfacing Cloud Function.

2.2 Related Work

The related work surveyed in this section provides architecture and experimental evaluation background to the research topic of this thesis. We start with research on evaluating persistent storage for IoT device management. The experimental setup, evaluation and technologies used is valuable to guiding our

experiments and evaluation. Our topic and experiments use the surveyed research as a starting point without duplicating it. Next, other research papers evaluating NoSQL databases against each other or against SQL are surveyed. These provide a methodology and benchmarking toolkit that is considered in the design of our experiments.

The next group of papers focuses on the specific technology used in our experiments. Firestore, MySQL and Spanner papers are surveyed and establish how performance is evaluated and the ability of each to scale. Spanner provides a physical storage layer for the Firestore service and is thus included. Finally, papers discussing approaches to augmenting MySQL such that it can scale are included. These provide background to the challenges faced with horizontally scaling a SQL database.

2.2.1 Performance Evaluation of IoT Data Management

Eyada et al. [17] is a paper focused evaluating database latency and size as they are applied to large volumes of IoT sensor data. Both MySQL and MongoDB (NoSQL, document-model) databases were evaluated while being hosted in AWS EC2 instances. The experiments varied the workload, compute resources and number of sensors. The resulting latency values fed in to a predictive model equation developed with both linear and non-linear regression methods. The results favored MongoDB for latency while MySQL latency did not grow as dramatically with increased sensor data.

2.2.2 Benchmarking with YCSB

Pandey [22] provides a comparison of relational and non-relational database platforms through comparing MySQL and MongoDB. The benchmark suite Yahoo! Cloud Serving Benchmark (YCSB) varies workloads on each database and captures latency and throughput measurements. All the measurements favored the MongoDB solution. However, the authors acknowledge and discuss the lack of strict atomicity, consistency, isolation and durability (ACID) properties with non-relational as well as the missing join operations for queries. The authors further conclude that MongoDB outperforms MySQL in sharding, security, performance and availability.

Another paper discussing the NoSQL platforms is Khazaei et al. [13]. The authors explore some popular NoSQL databases and describe the characteristics of this solution. The discussion includes the loosening of consistency, availability, partition tolerance (CAP) theorem and the resulting basically available, soft-state, eventually consistent (BASE) systems. The authors further compare multiple benchmarking suites including YCSB, PixMix, GRIDMix, CALDA, etc. and conclude with choosing YCSB for its flexibility.

2.2.3 Google Firestore

The first paper written on Firestore is Kesavan et al. [25] and is developed by a group of researchers within Google. This paper digs in to the architecture of Google's Firestore service, specifically addressing how it scales across many nodes, provides real-time notification capability, is easy to use and provides a pay-as-you-go billing model. The authors present benchmarking data that

shows little increase in query latency given large numbers of stored documents and increased document size and data shape. They conclude Firestore provides a convenient ecosystem with low barrier of entry for developers to rapidly prototype, deploy, iterate and maintain applications.

2.2.4 Oracle MySQL

MySQL is widely used by the majority of small and medium-sized applications [5]. It was initially released in 1995 and is developed by Oracle Corporation. In Bannon et al. [24], the authors dig into the MySQL server architecture and discuss the relational database management system (RDBMS) in generic terms that can apply to other SQL systems. The RDMBS is subdivided into application, logical and physical layer software that runs directly on the node containing physical storage. SQL requests are processed locally through a pre-compiler to extract the SQL statement, a query parser to convert to a parse tree structure with stored indices used where possible.

2.2.5 Sharding a Relational Database

Create, read, update and delete (CRUD) operations are all supported through a Structured Query Language (SQL) interface. These operations must adhere to a pre-defined schema that defines each column of the target table. Schematization provides a sanity check to ensure that incoming row insertions are conforming to the desired data shape and column types. Note that under certain circumstances, the forced structure may be considered a restriction. Any changes to a table's schema requires the entire table to be migrated to the new shape. Such

a restriction may provide development hurdles as an application is rolled out iteratively and persistent data requirements evolve.

The support for SQL query processing is a benefit. The language has been in existence since the early 1970s [3] leading to a large user support group, extensive documentation and teaching materials [28]. A huge advantage of the language is the ability to combine multiple inequality predicates within a single query. For instance, a *select* statement may contain multiple columns with inequality conditions. The statement is executed within a single operation and allows a response set to be minimized to only the desired matching records. We will see later how querying multiple conditions is significant to the energy auction. SQL also supports *join*. Queries to different tables are combined within a single operation. The *join* allows data referenced through a relation to be accessed from their respective tables. However, this feature does cause additional latency for read operations since the RDBMS must assemble the response from multiple tables.

Although the Applications and Interfaces layer in Figure 3.3 may exist on an external client, the remaining server architecture is largely monolith and designed to run alongside the physical storage within a single node. As table sizes and concurrent requests grow, the owning organization will typically upgrade the hosting hardware: i.e. faster processors, more memory and extra disk storage [5]. These upgrades do have limitations though, and at some point the organization must consider partitioning the large tables into shards and redistributing to other nodes. The redistribution requires complex design and administration. Relations need to be updated, application code may have to change, and

ACID transaction properties become more difficult to guarantee. Additionally, downtime from the user is frequently required to facilitate the sharding.

Dong and Li [5] start their research by identifying the issue that many organizations have built persistent storage around a relational database management system (RDBMS) such as MySQL. As information volume has seen an explosive increase, these systems prove difficult to expand. To address this, most organizations upgrade the database server hardware, migrate to a NoSQL or perform sharding such that the smaller partitions may be distributed.

The authors implement a novel middleware application that allows multiple MySQL databases to interact as a single, distributed entity. A MySQL interface is exported to external clients. Incoming queries are parsed and executed locally, then sent to the appropriate node across the distributed cluster of MySQL servers. A sharding algorithm is also run within the middleware allowing slices of data to be sent to the appropriate data nodes. The algorithm is defined through a set of rules within the middleware configuration files. Direct and semi-direct table-join strategies are implemented in the master node to support queries across the distributed nodes. After performing some functional tests, the authors conclude that their distributed database has slower response time than a single database being queried for the same data. However, the advantage of their system is evident with massive datasets that cannot fit within a single MySQL server.

Yadav and Rahut [31] describes how Meta redesigned their MySQL datastore replication protocol to use a modified version of Raft [21] instead of the traditional semi-synchronous replication. Raft is a consensus algorithm utilizing an

elected leader. It imposes restrictions that only servers with the most up-to-date data can become leaders. Data is sharded into many MySQL databases utilizing a primary and many replicas. Raft enables the control-plane and data-plane operations to be part of the same replicated log. Membership and leadership is moved inside the MySQL server, resulting in provable correctness during promotions and membership changes. This paradigm does not make sharding any easier, but rather optimizes the replication across multiple existing MySQL replicated databases.

2.2.6 Google Spanner

Although Spanner is not used directly for the experiments in this thesis, it is relevant since it provides the storage engine to Firestore. Corbett et al. [15] is a collaboration of 23 authors within Google to discuss the mechanism and benefits of Spanner's distributed database service. The authors describe the architecture with a focus on the distributed storage and timestamp management. They claim the database is "semi-relational" with schematized tables, yet still supports an SQL query interface. Below this layer, data is physically stored as key-value associated pairs across a large distributed network defined by a universe master, multiple zone masters and span servers. The paper presents benchmark results for commit time of 1 to 200 participants (ranging from 14.6 to 122.5 milliseconds). They conclude with a plug for the TrueTime timestamp API that is the "linchpin" of Spanner's advanced feature set.

Chapter 3

System Architecture

3.1 Firestore

The first set of experiments is centered on a NoSQL, non-relational database. Firestore was chosen for its flexible billing structure and serverless architecture. Firestore utilizes a document model [20] within its DBMS. The first benefit of the document model is being schema-free. Documents may be inserted with unstructured data allowing changes in data shape (i.e. different fields and types) without requiring a migration of the whole database. This is especially helpful when an application such as the energy auction is first ramping up and the persistent data requirements may not be fully understood initially. Thus, a schema-free paradigm allows incremental changes to what is persisted as the application matures. Schema-free contributes to backwards compatibility and database performance since it does not need to be taken offline to perform migrations.

Another benefit of the document model is horizontal scaling. Since documents exist independently of each other, it is possible to distribute a collection across multiple physical compute nodes. As document number and access traffic increase, this architecture allows for easy scaling with additional nodes. The scaling is nearly linear and utilizes distributed hash tables (DHT) that organize

[key, value] pairs into storage buckets [23]. These buckets may span unlimited server nodes.

Figure 3.1 shows a high level overview of Firestore's stack. Client access is provided through software development kit (SDK) libraries that come in two flavors. The "Server" SDK is trusted, and used in applications running within a Google Cloud service such as a Compute VM or Cloud Function. It requires a privileged environment, foregoes authentication and provides automatic retries with backoff.

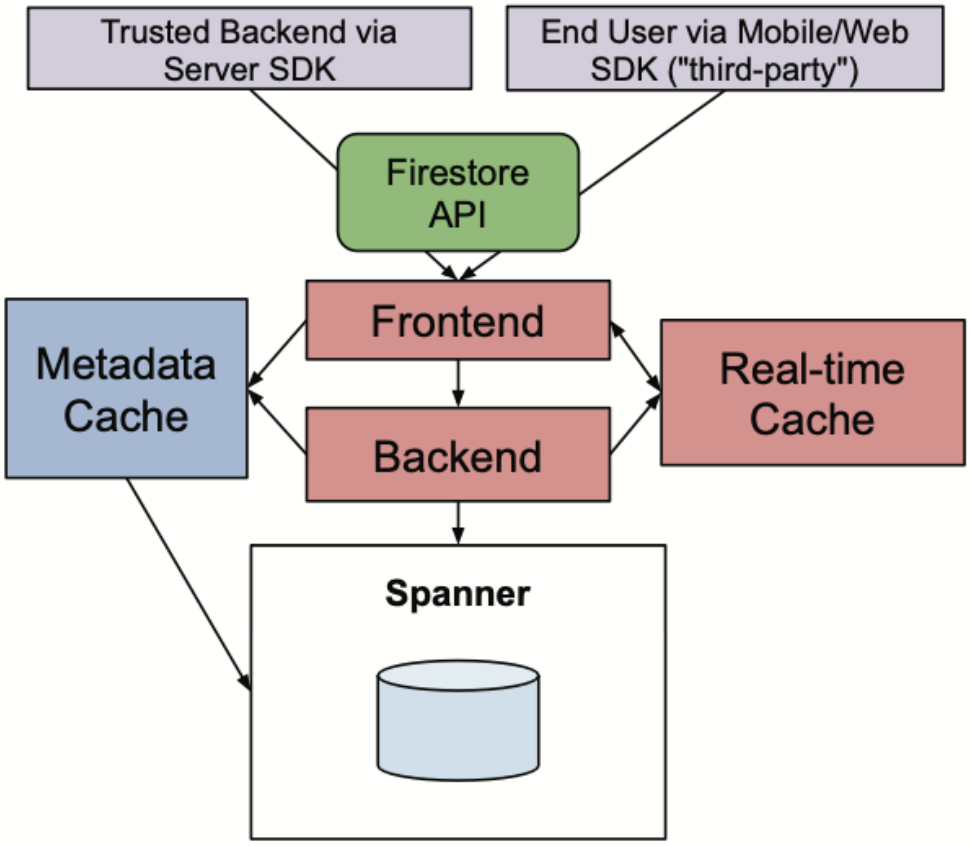

FIGURE 3.1: Simplified diagram of the Firestore stack [25].

The second SDK is the "Mobile and Web", which is intended for untrusted third-party devices. Both flavors of the SDK abstract connection to the Firestore API. They support blind writes and transactional writes based on optimistic concurrency control while connected. Queries are multiplexed over the same long-lived connection to the Frontend task. The Frontend tasks live in the same region as the database. This region co-location means that client SDK requests arrive at the closest-to-the-user Google point of presence, then get routed to the Frontend task after the database location is fetched from the Firestore metadata.

After a Frontend task receives the SDK request, it sends a remote procedure call (RPC) to a Query Matcher. Figure 3.2 expands on the Real-time Cache block from Figure 3.1 and shows the flow from multiple client users. The checking of timestamps in the In-memory ChangeLog ensures consistency with only valid commits being forwarded to the Backend task.

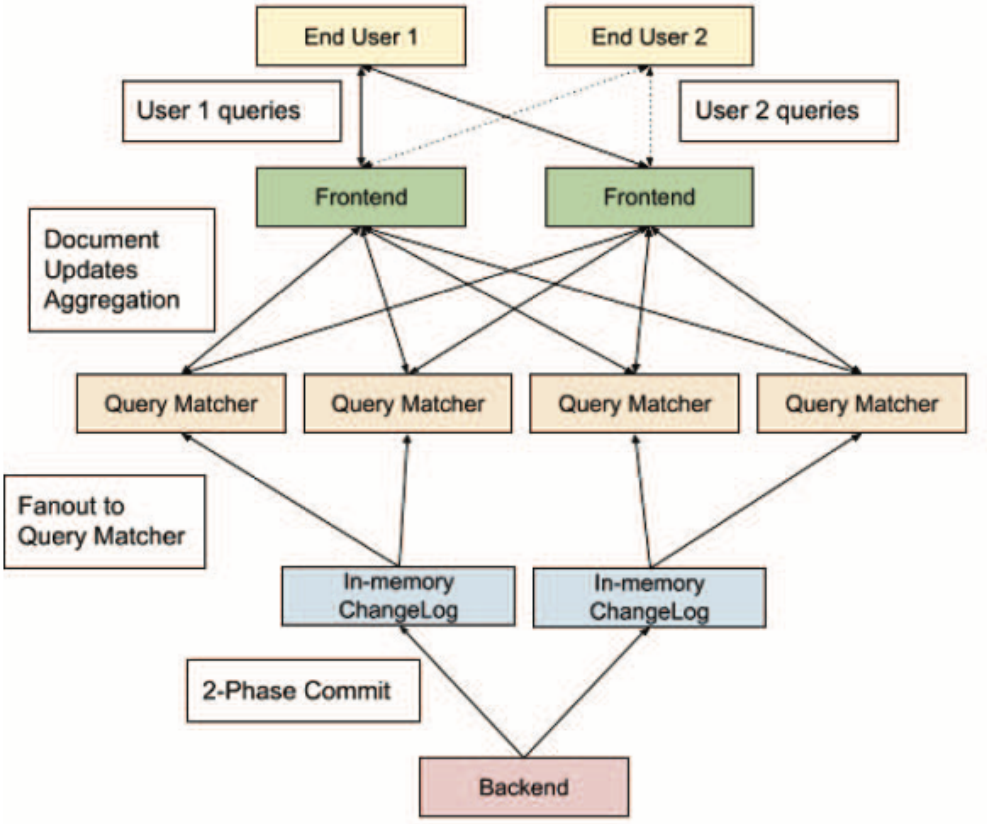

FIGURE 3.2: Query processing through the Real-time Cache [25].

Firestore executes all queries using secondary indices. Each field in a document automatically generates an ascending and descending ordered index, all on a per-collection basis. The automatic indexing may be excluded on a specific field if configured. Write operations are slightly more expensive latency-wise than read operations due to the need for updating multiple indices.

Firestore's database storage engine is Google Cloud Spanner [15]. Each Firestore database maps to a directory within some number of pre-initialized Spanner databases in the corresponding region. Each directory has two tables, *Entities* and *IndexEntries* containing the Firestore database data. Firestore documents are stored in the *Entities* table. The table contains one document per row, with the contents (up to 1 Megabyte) in a single column and the document key as the primary ID for the row. Each index is stored within a single row of *IndexEntries*. To handle distribution, load balancing and scaling, Spanner performs automatic splitting and merging of rows into *tablets*. This sharding [2] process, similar to those performed by other relational database management systems (RDBMS), allows for flexible horizontal scaling across an unlimited cluster of nodes.

3.2 MySQL

MySQL is a relational database since tables are structured as mathematical relations. The tables also support linking stored data objects between them (also called relations), i.e. a data column in one table may point to a record in another table. These relations support the normalization of data: a data item exists only in one table yet may be referenced repeatedly from other tables in a one-to-many relationship. Data normalization provides two main benefits, the first is the database may take less physical storage, since data duplication is avoided, and second, write operations may require less latency, since new data is only written to a single table and then referenced elsewhere if needed.

Another benefit of the RDBMS model is the strict adherence to the ACID

properties of a transaction. These are atomicity, consistency, isolation and durability. The MySQL Transaction Management layer allows data manipulation operations that ensure the database does not have the results of a partial operation [22]. Figure 3.3 graphically illustrates Transaction Management sitting between the Query Processor and Storage Management.

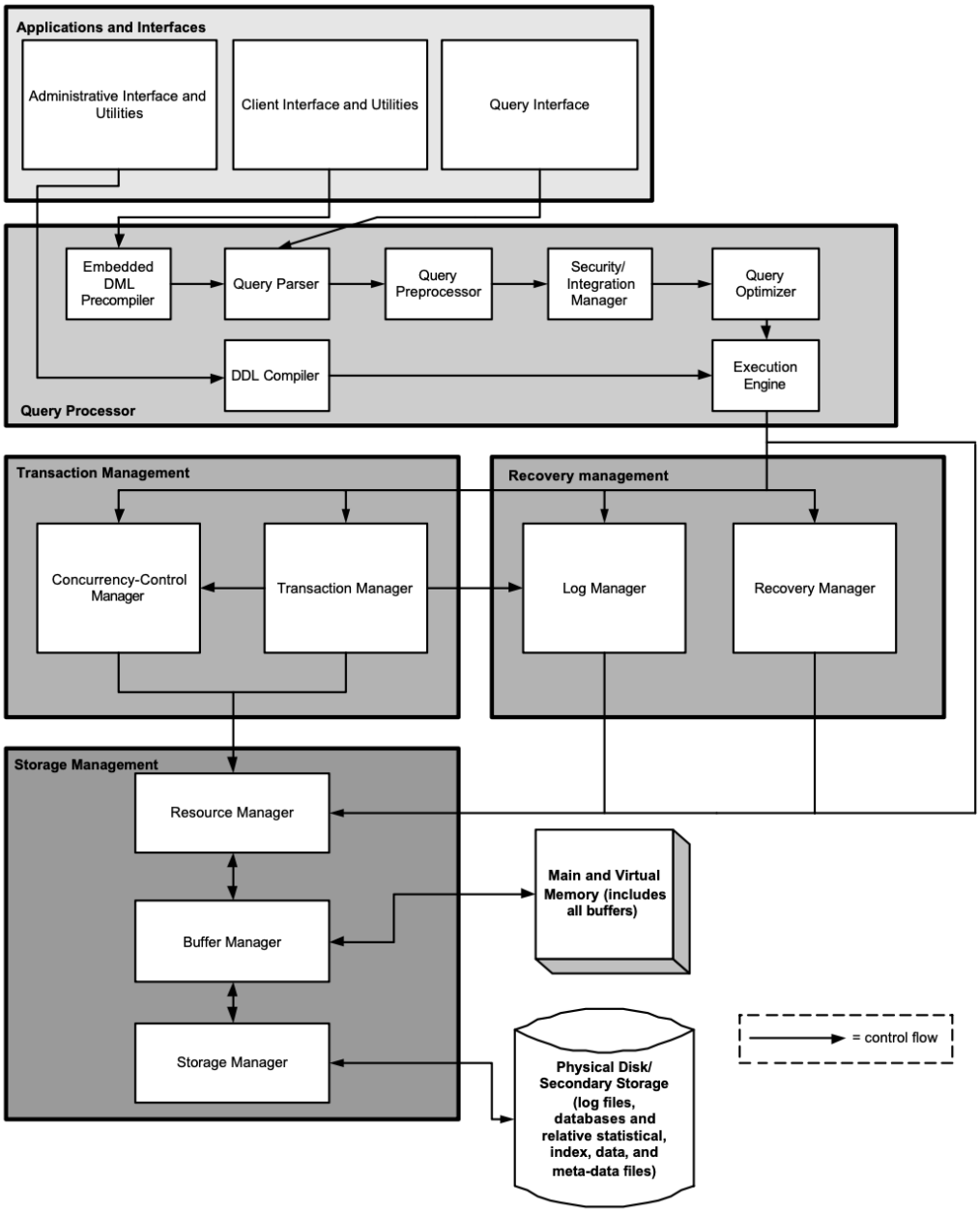

FIGURE 3.3: Detailed heterogeneous conceptual MySQL architecture from Bannon et al. [24].

Chapter 4

Experiments

The experiments provide a comparative benchmark of read and write performance between relational and non-relational database engines. In all cases, n represents the number of documents in a non-relational collection or the records in a relational table. The set of documents or records in the response is represented by r. Rather than using a third-party benchmarking tool, these experiments utilize a custom client performing read and write operations that mimic workflows in the energy auction. Registering a new IoT device, reading its persisted data and querying for specific combinations of fields are shown in Figure 1.1 and are covered in these experiments. In addition to performing operations to mimic the auction workflow, we also hit the databases with high-volume operations to determine the limits of the read data rate and the read and write operation rate. The aim for the experiments is to show the relationship between query latency and the number of database objects as well as the effect of multiple-condition queries.

4.1 Non-Relational, NoSQL Database

The Cloud Firestore database is divided into 6 collections spanning 10^1 to 10^6 documents. These documents represent mock data for the energy auction and

the different collection sizes are used to observe the effect of size on latency. Each document is a sample registration of a charger device with *name*, *id* number, *latitude*, *longitude*, *address* and charger *type*. All have a unique key string. Each mock data set is generated with a random generator algorithm that cycles each field between a range of accepted values. Write performance is tested through a single write of a mock charger registration to each collection (i.e. registering a new charger) followed by rapid sequential writes of all documents. These experiments measure both the time to execute a single write and how many write operations per second are supported by the database. The read performance is measured through a single document read of each collection followed by a rapid sequential read of all documents. Time to execute a single read operation is measured followed by the attainable read operations per second. Next, the client performs a bulk read of all documents in each collection. The time to execute provides the data rate which can be read.

Pattern querying is addressed next after measuring the combinations of reading and writing documents. A specific query pattern is defined with equality conditions for the *name* and *type* fields and inequality ranges for both *latitude* and *longitude*. Three queries are executed against each collection:

1. Read all documents and search for matches.
2. Read documents matching a single field range query (*latitude*) and search the subset for remaining matches.
3. Read documents matching a composite field query (with *latitude* and *type*) and search the subset for remaining matches.

4.1.1 Reads

For the first experiment, a single document was read from each collection using the document key. Firebase stores every document with a unique key identifier. Note the key is a primary index, separate from all other fields which are secondary indices. The 6 collection sizes (n) are:

1. n = 10 documents
2. n = 100 documents
3. n = 1000 documents
4. n = 10000 documents
5. n = 100000 documents
6. n = 1000000 documents

For each, the document key *1_11899NW118thAve* was used to perform a single read operation. Figure 4.1 shows the time to complete each read operation is reasonably consistent across the collection size range. This behavior is expected given the distributed hash table (DHT) lookup of the primary key. The number of nodes the collection spans over is handled transparently by Firestore. The read and write experiments were performed three times (minimum) with the first dataset being used provided there were not large discrepancies (greater than 10% difference in latency) between datasets. Once the data is retrieved by the client, it is viewed and then discarded. Persistently storing the retrieved data on the client is not included in the latency calculation.

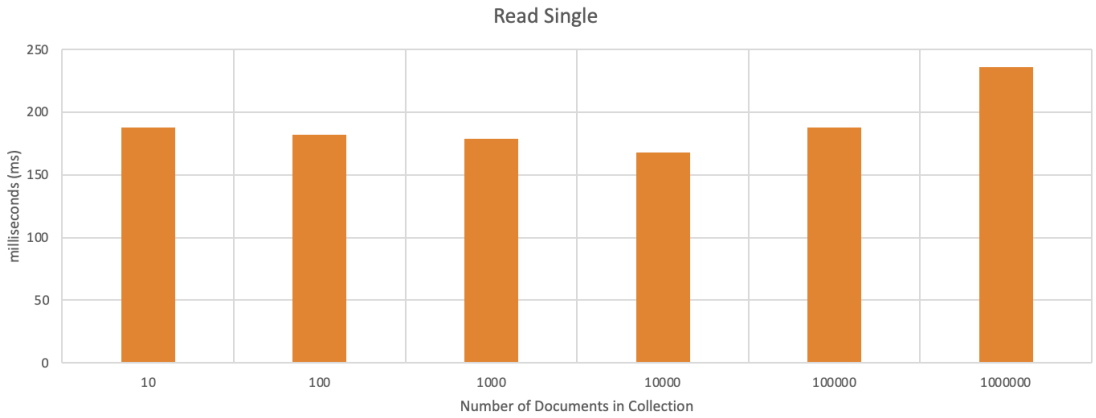

FIGURE 4.1: A single read operation performed against each of the 6 collections. Time to complete the operation is shown in milliseconds (ms).

The next experiment involves iteratively reading every document in each collection. The *id* field, a secondary index, is used to perform a simple read operation. The *id* field is a sequential integer assigned to each document. It has no direct meaning to the charger represented in the document but allows for an easy sequential query of all documents in the collection. Note that this field is automatically indexed by the Firestore service to increase performance of their query engine.

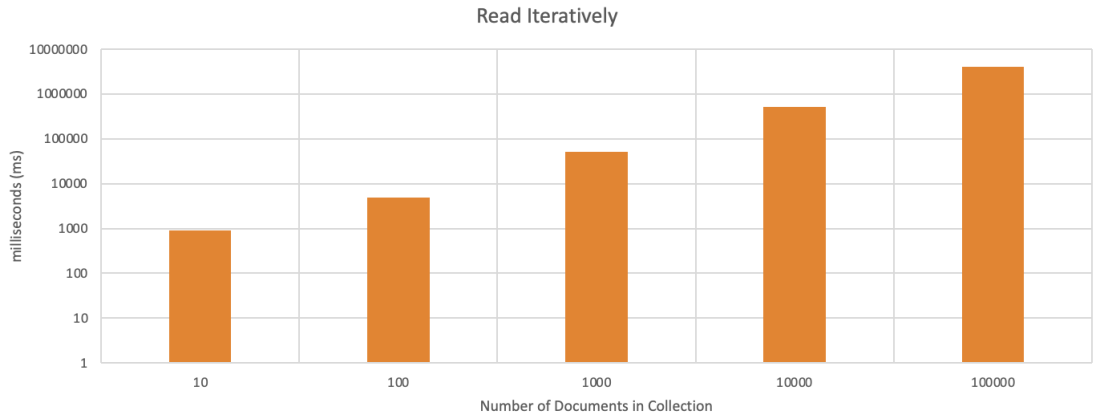

FIGURE 4.2: An iterative read operation performed against five Firestore collections using a secondary index. Each unique *id* field is successively queried. Time to complete all the operations is shown in milliseconds (ms) on a logarithmic scale.

Figure 4.2 shows the time taken to iteratively read all documents. Sequential reads like these do not directly represent a user workflow in the auction, but do allow us to characterize the read operation rate of the database (shown in Table 4.1). Time complexity consists of the query time for each *id* field across the n documents in the collection. Thus, the total time is

$$T_C(n) = O(n \cdot O_D(n)) \tag{4.1}$$

where

D: A single read operation via an *id* query.

C: Iteration over the entire collection.

Since the query of each document utilizes a pre-generated, secondary index,

the query engine performs a binary search to quickly match the *id* value in the index. This binary search has worst case time complexity of $O(n) = \log n$. Substituting this into Equation 4.1, we can simplify to:

$$T(n) = O(n \cdot \log n) \tag{4.2}$$

Or approximately linear time. Note that $r = 1 \forall n$ since every iteration returns exactly 1 document.

The final read experiment involves a read of all documents in each collection. The operation does not use a primary or secondary index. Figure 4.3 shows the time taken to read all documents by collection. In this scenario, $r = n \forall n$ since all documents in the collection are always returned.

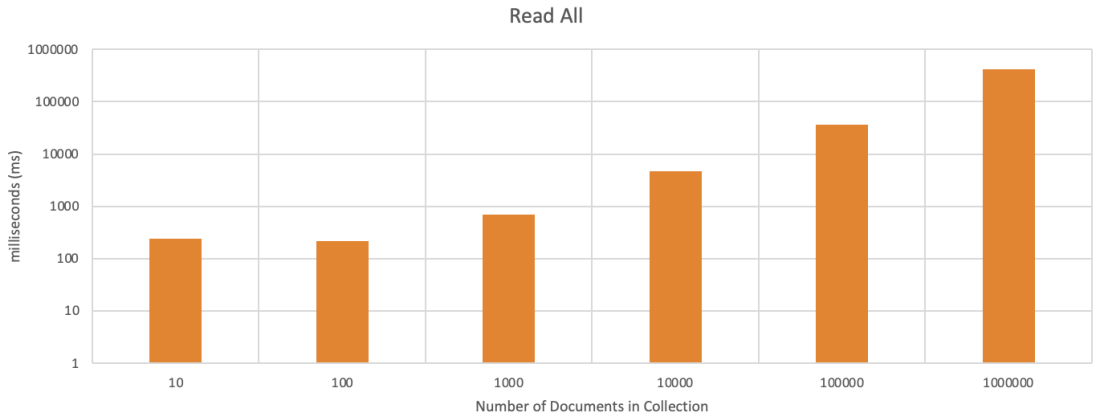

FIGURE 4.3: A read all operation performed against each of the 6 collections. All documents are read in a single operation. Time to complete all the operations is shown in milliseconds (ms) on a logarithmic scale.

The time to read all documents is directly proportional to the number of

documents returned. A bulk query such as this is not in any user workflow but does allow the characterization of the data rate as seen in Table 4.1. Time complexity for this operation is linear and represented by Equation 4.3.

$$T(n) = O(n) \tag{4.3}$$

We finish the read analysis with the performance metrics shown in Table 4.1. The time in milliseconds to perform a single read operation with a known key (primary index) is taken straight from Figure 4.1. The operations per second are calculated by dividing the number of iterative operations in Figure 4.2 by the time to complete. The kilobytes per second data rate uses an average document size of 233 bytes, multiplies by the number of documents and divides by the time taken to complete a read all operation as shown in Figure 4.3.

TABLE 4.1: A summary of all read metrics. The first column represents the number of documents in the collection. The second is the time taken for a single read operation with a known key. The third is the operations performed per second while iteratively reading all documents via a secondary key. The fourth column represents data rate during a read-all operation.

	n	**ms/oper**	**ops/s**	**kB/s**
	10	188	10.9	9.7
	100	182	20.6	109.4
	1,000	179	19.2	340.6
	10,000	168	19.3	503.9
	100,000	188	24.8	644.3
	1,000,000	236		554.7
Mean		190.2	19.0	360.4

4.1.2 Writes

For the next set of experiments, we will focus on characterizing the write operations, both create and update. These largely mirror Section 4.1.1 and much of the background explanation applies. The first experiment performs a single update operation on each collection. The update changes all fields in an existing document. A known key, *1_11899NW118thAve*, is used in every case. Note that for all create operations, a previously generated and randomized set of mock data is used to represent each charger device.

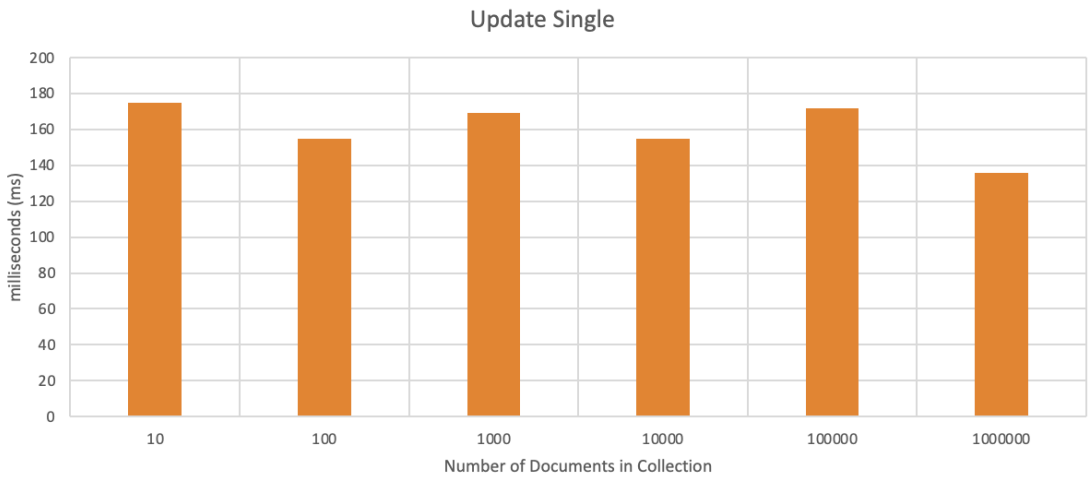

FIGURE 4.4: A single update operation performed against each of the 6 collections. Time to complete the operation is shown in milliseconds (ms).

Figure 4.4 shows the time in milliseconds for the operations to complete. As with the reads, the single writes complete in consistent time regardless of the collection size. Next, iteratively writing all documents in the collections are shown in Figure 4.5. As with reads, this scenario does not represent a user workflow, but allows us to characterize the write operation rate as shown in Table 4.2. A high write operation rate is not critical for the energy auction but is a useful metric when comparing the NoSQL vs SQL. Time complexity for all n operations to complete follow that of the read scenario where $T(n) = O(n \cdot \log n)$. It is approximately linear time. Note that a write-all operation is not possible through this client interface and is not tested here.

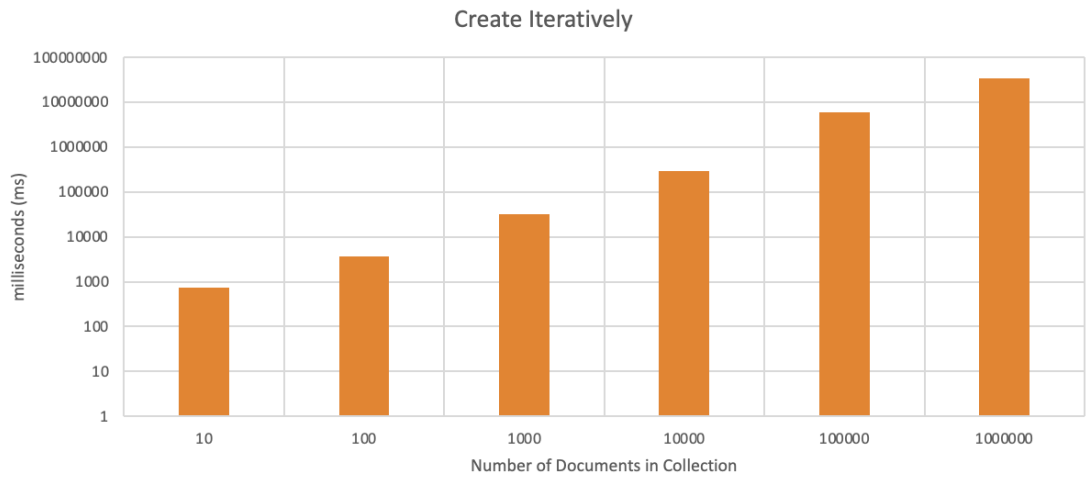

FIGURE 4.5: An iterative create operation performed against each of the 6 collections. All n documents are created in sequence. Time to complete all the operations is shown in milliseconds (ms) on a logarithmic scale.

TABLE 4.2: A summary of all write metrics. The first column represents the number of documents in the collection. The second is the time taken for a single update operation with a known key. The third is the operations performed per second while iteratively creating all documents.

	n	ms/oper	ops/s
	10	175	13.8
	100	155	27.2
	1,000	169	31.6
	10,000	155	33.8
	100,000	172	16.5
	1,000,000	136	29.4
Mean		160.3	25.4

Table 4.2 summarizes the milliseconds per write operation as shown in Figure 4.4 and the write operations per second that are possible as shown in Figure 4.5. Although write results largely mirrored the read, we (surprisingly) see a slight performance improvement.

4.1.3 Reads with Conditions

The set of experiments that most mirrors real-world scenarios are the querying of complex field patterns. Firestore does not permit multiple range (inequality condition) queries of secondary indices. Therefore, to read the documents that match all the query conditions, some combination of querying the database

and data reduction at the client must be performed. For these experiments, the following query conditions are used:

```
name: Howard
latitude: min 47.5 and max 48.0
longitude: min -122.5 and max -122.1
type: level2
```

The first of the query experiments reads all documents without a query. A client-side data reduction narrows down the documents in each collection that match the given query parameters. This full data reduction is the most brute-force method and is shown here for comparison. Since we are always getting every document, $r = n$ in each case.

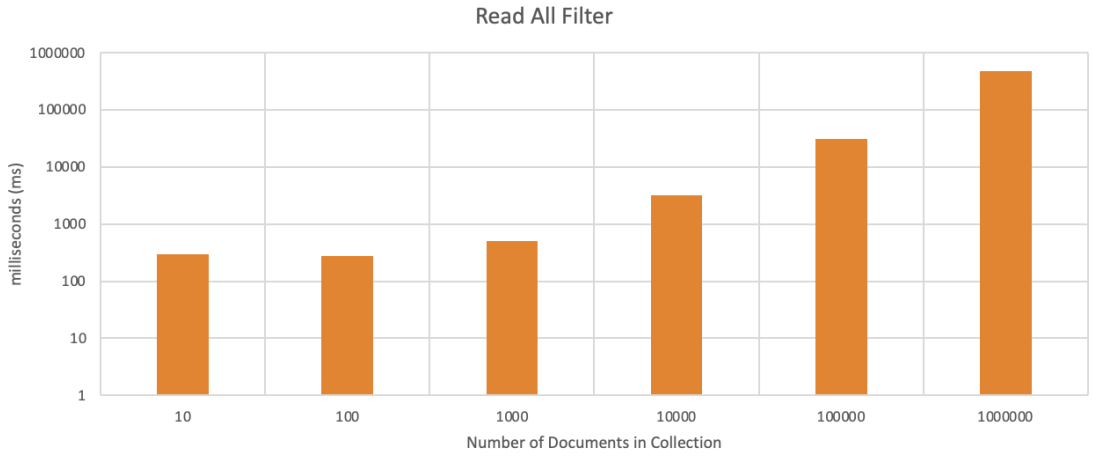

FIGURE 4.6: A read all performed against each of the 6 collections. All documents are read in a single operation, then reduced locally to those matching the query conditions. Time to complete all the operations is shown in milliseconds (ms) on a logarithmic scale.

Figure 4.6 shows the time to read all documents and reduce matched very closely with the time to read all from Figure 4.3. The reduction time on the test machine (M1 processor) is negligible. Thus, the response set size r is the defining factor for time complexity which is linear time or $T(n) = O(n)$.

The next experiment performs a conditional query using the *latitude* field. It defines a range between the minimum and maximum values. The Firestore query engine can only handle a single field range, thus we query only with *latitude*. Firestore performs a binary search on the pre-generated *latitude* index and returns a set of documents (r) that must be searched locally on the client to match *name*, *type* and *longitude*. Figure 4.7 shows the time spent on the query and subsequent data reduction.

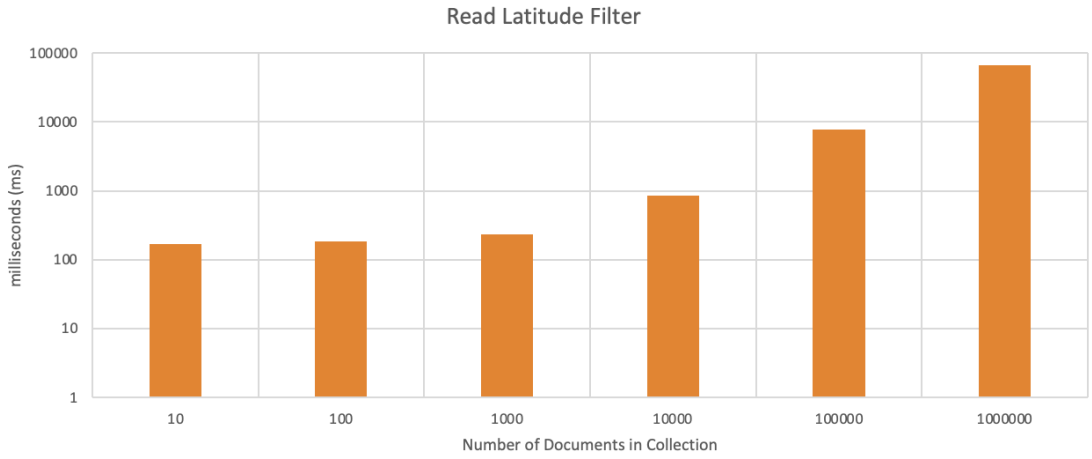

FIGURE 4.7: Query the *latitude* range in each of the 6 collections. All *latitude*-matching documents are read in a single operation, then reduced locally to those matching the non-*latitude* query conditions. Time to complete all the operations is shown in milliseconds (ms) on a logarithmic scale.

The final query experiment involves a composite index. Since each field has a corresponding index which is automatically generated, Firestore allows queries to combine multiple fields provided that they are equality (i.e. "==") conditions. Inequality (i.e. "<=") conditions, also known as range queries, cannot be combined. This type of multi-field range query is very relevant to the energy auction. A typical user workflow would have, at minimum, a *latitude, longitude* range and charger *type* as conditions of the query. Time to complete each query and reduction is shown in Figure 4.8.

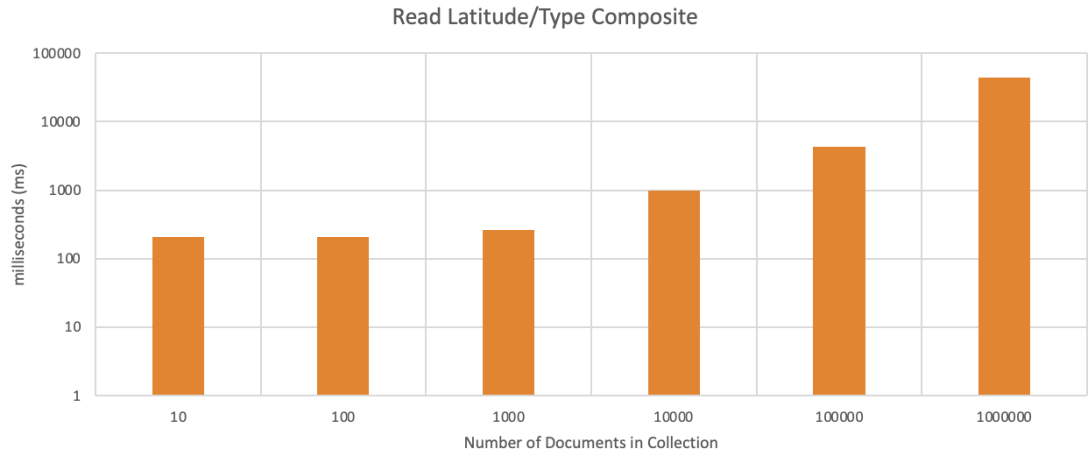

FIGURE 4.8: Query the *latitude-type* composite index in each of the 6 collections. All documents are read in a single operation, then reduced locally to those matching *longitude* and *name*. Time to complete all the operations is shown in milliseconds (ms) on a logarithmic scale.

To summarize the query possibilities, Figure 4.6 queries all documents in each collection and does a full data reduction (*latitude, longitude, name* and *type*) at the client. Figure 4.7 queries the *latitude* range and does a data reduction

for *longitude*, *name* and *type*. Finally, Figure 4.8 shows the query of a composite index (*latitude* and *type*) with a data reduction of *longitude* and *name*. Table 4.3 summarizes the latency times across these three query possibilities for all collections. Note that the *latitude* query provides the lowest latency up until a collection size of 10,000. Beyond that, querying the composite index *latitude-type* completes with the lowest latency.

TABLE 4.3: A summary of querying all documents, *latitude* only and the *latitude-type* composite index. In each case, a data reduction is performed at the client side to match the *latitude, longitude, name* and *type*.

n	**Query All**	**Query Latitude**	**Query Latitude-Type**
10	300	171	210
100	273	182	207
1,000	508	234	264
10,000	3213	855	995
100,000	31425	7718	4330
1,000,000	469787	66763	44726

4.2 Relational, SQL Database

The next set of experiments focuses on the SQL database and how performance differs from the NoSQL. The terminology we use to describe the experiments changes slightly from the NoSQL scenario in Section 4.1. "Collections" and "documents" are now replaced with "tables" and "records (or rows)". "Fields"

are replaced with "columns". The same mock data from our NoSQL experiments is inserted into 6 tables that span 10^1 to 10^6 records. The largest difference in the experimental procedure is queries always return exactly the matching results. There is no need for the scenario described in Section 4.1.3 where a larger than needed result set r is returned and further reduced client-side to the matching records.

4.2.1 Reads and Writes

The single and sequential read/write operations show similar results to Firestore in Section 4.1. A summary of the read metrics, Table 4.4, shows that the milliseconds per operation and the operations per second remain fairly constant with the increasing n. However, data rate increases proportionally to the increasing n due to the almost constant read time.

TABLE 4.4: A summary of time per operation, operations per second and data rate in kilobytes for each of the 6 table sizes. The first column is the number of records per table. The second column is the average time to complete the iterative reads across all records. The third column represents how many read operations per second were performed during the iterative reads. The fourth column shows the data rate during a bulk read of all records.

	n	**ms/oper**	**ops/s**	**kB/s**
	10	84	39	26
	100	82	55	274
	1,000	105	56	1,879
	10,000	86	63	12,944
	100,000	90	63	48,240
	1,000,000	106	49	624,665
Mean		92	54	114,672

TABLE 4.5: A summary of time per operation and operations per second for each of the 6 table sizes. The first column is the number of records per table. The second column is the average time to complete the iterative writes across all records. The third column represents how many write operations per second were performed during the iterative writes.

	n	**ms/oper**	**ops/s**
	10	101	26
	100	80	52
	1,000	88	49
	10,000	95	53
	100,000	96	54
	1,000,000	91	39
Mean		92	46

We complete the summary of write metrics through Table 4.5. The average was approximately 46 operations per second regardless of the table size n. With both the time per operation and the operations per second approximately constant, we conclude writes to a SQL, relational database may be performed in constant time or $T(n) = O(1)$.

4.2.2 Reads with Conditions

The final set of benchmarks are querying with the same conditions as listed in Section 4.1.3. These results are the most significant since they show the scenario where the SQL database outperforms the NoSQL. This experiment represents

the auction scenario where a user is querying a *latitude, longitude* range along with charger *type* and customer *name* as conditions. Figure 4.9 shows a very slight increase in time to completion as the table size n increases. The increase in time can be attributed to the larger response set r. Note that, unlike with the non-relational multi-field range queries, only the matching records are returned. The query itself performs in constant time, $T(n) = O(1)$, which is a performance improvement over the non-relational. As the response set grows larger (i.e. $n = 10^6$), the execution time does increase slightly. The increase is proportional to r (r grows linearly with n) but is kept minimal since only records matching the query conditions are returned.

FIGURE 4.9: Query with name=Howard, type=level2, 47.5 $\leq$ latitude $\leq$ 48.0, -122.5 $\leq$ longitude $\leq$ -122.1. Time to complete all the operations shown in milliseconds (ms) across all 6 tables.

Chapter 5

Discussion and Future Work

In Section 1, we state that the evaluation of the persistent storage solution would be based on:

1. Latency performance.
2. Cost.
3. Flexibility and scalability.

These criteria are now discussed for both the Firebase and MySQL solutions in the following sections.

5.1 Latency

In this section, we show an analysis to estimate latency based on dataset size n, and response set size r. The analysis is performed on experimental latency results from both Firestore and MySQL databases, with the aim of providing a predictive model for each. The models estimate latency in milliseconds.

The evaluation needs to find a relationship between three variables: l (latency), n (data size) and r (response size). Throughput is not considered in this relationship. The dependent variable l is related to the independent n and r. A Multiple Linear Regression takes a general form of:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \tag{5.1}$$

Y is the output or dependent variable. The X terms represent the corresponding input or independent variables. Each predictor (input) has a corresponding slope coefficient β. The first β (β_0) represents the intercept constant. In our case, we can restate this equation more specifically as:

$$l = \beta_0 + \beta_1 n + \beta_2 r \tag{5.2}$$

The determiner method [17, 4, 16] is used to solve Equation 5.2.

$$\begin{vmatrix} l & n & r & 1 \\ \sum_{i=1}^{n} l & \sum_{i=1}^{n} n & \sum_{i=1}^{n} r & n \\ \sum_{i=1}^{n} nl & \sum_{i=1}^{n} n^2 & \sum_{i=1}^{n} nr & \sum_{i=1}^{n} l \\ \sum_{i=1}^{n} rl & \sum_{i=1}^{n} nr & \sum_{i=1}^{n} r^2 & \sum_{i=1}^{n} r \end{vmatrix} = 0 \tag{5.3}$$

Rather than solve Equation 5.3 by hand, we use a data analysis tool [18] to calculate the β coefficients on the following data.

5.1.1 Firestore Analysis

We will only focus on the read operation utilizing a *latitude* range condition shown in Table 5.1. The other fields such as *longitude*, *name* and *type* must be filtered at the client side and are included in the latency values. This is the most applicable scenario in the energy auction. Create, update and delete operations are not as relevant to the auction performance and user experience.

TABLE 5.1: Firestore latency for read operations utilizing *latitude* range condition from Figure 4.7. Both n and r are predictor variables and l is in milliseconds.

Latency (l)	**Documents (n)**	**Results (r)**
171	10	1
182	100	3
234	1000	18
855	10000	183
7718	100000	2020
66763	1000000	7745

Multiple Linear Regression analysis on Table 5.1 yields $\beta_0 = 154.924$, $\beta_1 = 0.061$ and $\beta_2 = 0.722$. Thus, our predictive model is:

$$l = 154.924 + 0.061n + 0.722r \tag{5.4}$$

The standard error for these coefficients are: $SE_{\beta_0} = 14.461$, $SE_{\beta_1} = 0.0002$ and $SE_{\beta_2} = 0.025$. To relate Equation 5.4 back to predicting a query's latency performance, note that the number of documents in a collection n only contributes a coefficient of 0.061. The number of documents in the query response r contributes 0.722. The factor of response to collection documents is $\frac{r}{n} = \frac{0.722}{0.061} = 11.836$. Thus, we conclude that the response size is approximately $12\times$ more significant in impacting latency performance than the size of the original collection. Our optimization efforts should therefore be focused on reducing the

response set to reflect the matched documents as much as possible.

5.1.2 MySQL Analysis

Now we do the same for the MySQL read operations querying a specific combination of conditions. In this scenario, all four conditions are added to the query such that only matching records are returned. The four-condition MySQL query contrasts with the Firestore results from Section 5.1.1 above where the query engine could only support a single field range condition.

TABLE 5.2: MySQL read operation latency while querying *latitude, longitude* range and *name, type* equality condition from Figure 4.9. Both n and r are predictor variables and l is in milliseconds.

Latency (l)	**Records (n)**	**Results (r)**
101	10	1
80	100	0
84	1000	2
122	10000	32
156	100000	403
905	1000000	3874

Performing the same analysis on Table 5.2 results in β_0 = 92.467, β_1 = 0.006 and β_2 = -1.332. Our predictive model is

$$l = 92.467 + 0.006n - 1.332r \tag{5.5}$$

Equation 5.5 shows us that the number of records in the table has a small effect on latency. There is a slight (0.006 factor) increase in latency given an increase in n. However, this small latency increase does need to be qualified with the fact that the table can only grow to a finite size. Beyond that, either hardware resources must increase or sharding must partition the table across other nodes. Note the negative value of the r coefficient, indicating its impact is even less than the total records in the table n.

5.2 Cost

To objectively compare the cost of both solutions, we focus on the worst-case scenario storing 10^6 charger data objects over one month. We calculate the cost to create, store and read the entire dataset. Each charger object create and read is performed in an individual operation. The average object size is 233 bytes, giving a stored data size of 233 MB. Here is the breakdown:

5.2.1 Firestore

Firestore only bills per use, i.e. per CRUD operation and per unit of storage [9]. Note that the experiment is performed in a single day, yet the service remains up for one month.

TABLE 5.3: Firestore cost estimate to create, store and read 10^6 documents (233 MB) in one month.

	Free quota per day	**Price per unit after**	**Unit**	**Total**
Document writes	20,000	$0.09	per 100,000	$0.882
Document reads	50,000	$0.03	per 100,000	$0.285
Document storage	1 GB	$0.15	GB/month	$0
Total				**$1.167**

As shown in Table 5.3, the total cost for a one-month experiment is $1.17. Note this cost is only incurred during use. CRUD operations are only billed after the free daily quota is exceeded. Data storage costs start incurring after the free quota, however, our 233 MB in this scenario is well below that for the initial charger registration. As the number of charging sessions increase, the storage requirements will also grow. Thus, we will likely see document storage charges that are beyond the free tier.

5.2.2 MySQL

Cloud SQL MySQL follows a different billing model. Costs are incurred for the dedicated compute along with the corresponding storage [11]. In our experiments, the service is provisioned with 1 vCPU, 614.4 MB memory and 10 GB SSD storage for one month.

TABLE 5.4: MySQL cost estimate to create, store and read 10^6 documents (233 MB) for one month.

	Price per unit	**Units**	**Total**
vCPU	$30.149	1	$30.149
Memory	$5.11/GB	0.614	$3.14
Storage	$0.17/GB	10	$1.70
Internet Ingress	free		
Internet Egress	$0.19/GB	0.23 GB	$0.04
Total			**$35.03**

Table 5.4 shows our costs to run this service for one month is $35.03. As we are provisioning a perpetual server (not serverless), costs are incurred regardless if the service is used or not. Note: as usage increases, Firestore's pay-per-use model will eventually surpass MySQL in cost. For example, writing the same 233 byte documents in Firestore at a rate of 500,000 per day and 1,000,000 reads will result in a $35.48 monthly cost.

5.3 Flexibility and Scalability

Analyzing flexibility and scalability is a more subjective discussion (without empirical data) based on the system architecture as detailed in Chapter 3. To recap, the NoSQL, non-relational, document-model implemented by Firestore brings the following benefits:

1. Stored documents are schemaless, allowing new fields and a changing data shape as development of the auction progresses.
2. Data is stored as a document object, referenced by a unique key. Stored document cannot have relations to each other. Thus, horizontal scaling can be easily performed by distributing documents across multiple nodes, zones and regions. The horizontal scaling can be done as the database and client demands grow without requiring any service outage.
3. Separating the management layer (DBMS) from the storage engine allows for easy implementation of a serverless offering. A dedicated, perpetual server is not needed. Firestore listens for client requests on a behind-the-scenes shared compute cluster and only charges per transaction.

In contrast, Cloud SQL MySQL deploys a perpetual (non-serverless) instance that is always consuming compute resources along with associated billing. Tables must define a schema, and any updates to this schema require a migration of the entire table. The migration will typically involve service downtime as well as developer effort. The migration can be avoided if new tables are added after a schema change. Finally, as the table grows in data size and number of records, distribution across multiple nodes requires a complex sharding process involving splitting the table into child objects and recreating all affected relations. This sharding will also typically involve downtime for the users.

5.4 Discussion

In this section, we summarize the significant results and compare. In Sections 4.1 and 4.2, we deliberately chose to ignore delete operations and focused on create, update and read. The reads were divided into single operations querying via a primary key and complex queries involving *latitude*, *longitude*, *name* and *type* conditions. In the Firestore case, only one of these conditions may be an

inequality (range). MySQL is able to process queries containing multiple range conditions. This restriction means that Firestore is forced to return a larger response set than needed and perform further data reduction on the client. Both database architectures have atomic guarantees providing concurrency control for charger reservations.

The create, update and delete operation performance is not as relevant to the energy auction. With only one participant performing any of these operations at a time, it is unlikely that a small increase in latency would affect the user experience. However, the read performance is very relevant. Querying small datasets n with small response sets r is fast regardless of the database used. In the larger datasets, we begin to see the differences between the NoSQL and SQL solutions.

To illustrate, take the use case of 1,000,000 chargers. The Firestore solution stores each as a document while MySQL is a record within a table. MySQL is able to query these 1 million records for our specific combination of conditions (two are inequality and the other two equality) and return a matched 3874 records in ≤ 1 second (see Table 5.2). Firestore queried the same data distributed as 1 million documents in a collection using only the *latitude* condition. It took 66 seconds to respond with 7745 documents then further reduce to match the *longitude, name and type* conditions to get the desired 3874 hits. MySQL, with its more sophisticated SQL query engine, clearly has an advantage here. However, as stated in Section 5.1.1, the number of documents in the Firestore response set is 12 times more impactful to the latency of the operation than the number of

documents in the database collection. Optimizing collections to allow single-field queries responses to more closely match all desired conditions will have a significant impact on read performance.

Table 4.1 shows, in our experimental setup, Firestore read operations were able to average 19 operations per second while transferring 360 kilobytes per second. Table 4.4 shows MySQL achieving 54 operations per second with 114,672 kilobytes per second respectively. Clearly, MySQL holds the advantage in read performance while averaged across all dataset sizes. Comparing write operations (create or update) from Table 4.2 and 4.5, we see that Firestore is able to sustain an average of 25 operations per second while MySQL can do 45 operations per second. Again, the advantage to MySQL. However, a significant factor to consider with MySQL is table relationships. Our experimental setup has records independent of each other with no relationships. Normalizing data fields such that they exist only in a single table with relationships to it can impact read performance as these relationships must be resolved and joined in the SQL query engine. Additionally, table size consumes server resources and has a finite limit. Very large tables may translate into degraded performance at the upper limit requiring hardware upgrades or sharding to other nodes. Even with the raw latency advantage demonstrated by MySQL in our experiments, Firestore is preferable for expanding the auction long-term as it is the more consistent solution without any known upper limit.

Cost is a huge factor as described in Section 5.2. Our month-long scenario creating, storing and reading 1 million data objects resulted in a Firestore estimated cost of $1.17 while MySQL costs $35.03. The NoSQL document-model

database lending itself to serverless operation shows a significant advantage in providing a pay-as-you-use service. In the limited usage scenario of this study, Firestore has the cost advantage.

5.5 Future Work

In Section 5.4, we discussed how Firestore has an advantage with cost, schemaless flexibility, horizontal scaling and more consistent latency as collections scale up. The two disadvantages are higher latency, especially with complex multi-range queries, and data safety from a missing fixed schema. Technology potentially mitigating these disadvantages within the energy auction is the focus for future work.

5.5.1 GraphQL

GraphQL (or GQL) provides a query language and runtime [12] that can wrap the backend data storage and exposes an API that can be called from the client application. The query language provides a more sophisticated engine that can return all desired data within a single GraphQL operation. The data aggregation is done through the definition of a query type. Querying this provides only what the client needs and implements a strong typing on data contained within each field.

The strong typing of both data shape and fields compensates for the data safety risk of using a schemaless database. With GraphQL queries, CRUD operations must match the type specified, thus avoiding the scenario where a document field is overwritten with an incorrect data type. Even with GraphQL as an

intermediate to Firestore, the same queries must eventually be made to the Firestore collection. GraphQL is only wrapping, resulting in the same latency times we encountered querying Firestore directly. However, GraphQL can act as an aggregator: pulling data from multiple Firestore collections simultaneously or implementing a hybrid model with Firestore and MySQL. Such a strategy provides a framework to distributing the energy auction data in such a manner that querying a single collection may no longer be a bottleneck. Brito et al. [7] discuss migration strategies. A potential research topic is to implement a GraphQL solution with the aim of reorganizing data reads and writes in such a way as to reduce the latency times.

Chapter 6

Conclusion

The peer-to-peer energy auction for EV charging is a hybrid of Edge devices and Cloud services. Within these services, persistent storage of charger registration and reservation data is critical. Details of each charger device must be stored along with user details, session history, billing information, user ratings, amenities and tourism information. A Cloud-hosted database is the top choice to persistently store the data, support performant queries and provide atomicity, consistency, isolation and durability (ACID) guarantees. The nature of the auction workflow causes the create, read, update and delete (CRUD) operations to not have equal requirements. For example: create and update operations for registrations and reservations are performed by a single user at a time, thus resulting in eventual consistency being acceptable. Per the consistency, availability and partition tolerance (CAP) theorem trade-off, the auction prioritizes availability and partition tolerance. However, read operation performance is critical as it involves multiple users querying the same dataset within a mobile setting. Additionally, these read operations must be able to support querying multiple range (i.e. inequality condition) fields simultaneously.

This study focuses on two popular database services provided by Google to solve the research problem of persistent storage for the energy auction. Firestore

is a NoSQL, document-model, non-relational and serverless database service while Cloud SQL MySQL is a perpetual server, table-based, relational and SQL. Each has its own features and characteristics which we evaluate based on the following criteria:

1. Latency performance.
2. Cost.
3. Flexibility and scalability.

The SQL database is the established solution but does have shortcomings in terms of cost and scalability. The cost and scalability concern leads to our research question of can the Firestore solution be used to support the energy auction framework? We dig into Firestore's architecture and identify these benefits:

1. Schemaless data storage provides the flexibility to change document content (data shape) without requiring the complexity and outage to migrate the entire table. This does raise data safety concerns since a schema is not enforced on incoming data, leading to possible data errors or security vulnerabilities.
2. Each document is stored as a data object, independent of the rest. This granularity without relationships means the physical storage can easily be spread out across multiple nodes, zones and regions. Horizontal scaling can take place without outage although it may require duplicating data (i.e. all users that charge at a single charger object).
3. The document-model allows for easy separation of the database management system (DBMS) and the storage engine (based on a relational database). The separation enables a serverless offering using shared compute and pay-per-transaction pricing.

The more established MySQL architecture exists as a monolithic set of processes that must exist on a single node. Any distribution requires a complex

sharding of the table(s) and an update of the relations. However, MySQL provides the following benefits:

1. The SQL query engine is powerful and can support multiple column range conditions within a single query. The response records are an exact match preventing the need to perform data reduction at the client.

2. Data can be normalized such that only a single instance exists in the database with multiple relations pointing to it. The relationships allow faster writes since data does not need to be written in multiple places as well as reducing physical storage size.

Experiments were conducted to benchmark the latency performance under different load scenarios. The experiments mimic the energy auction workflow from an Edge client. CRUD tests were run against 6 datasets increasing from 10^1 to 10^6 individual data structures, each representing a charger device. Write operations using the primary key averaged 160 ms on Firestore but only 92 ms on MySQL. The read operations with primary key averaged 190 and 92 ms respectively. Firestore was only capable of an averaged 19 operations/s while MySQL could achieve 54 operations/s. An additional read scenario ignored the primary key and instead queried with two inequality conditions (*latitude, longitude*) and two equality (*name, type*). This combination represents a typical workflow in the auction. For up to 10,000 chargers, both databases returned the matched results in less than 1 second. However, as the dataset size n grew beyond, Firestore

was hampered by the limitation that its query engine could only support a single inequality at a time. Thus, we must query for *latitude*, return a much larger-than-necessary response set r and do a client-side data reduction. Since charger queries are typically location-based, the *latitude* field makes the most sense as the query starting point (instead of *name* or *type*). The worst case dataset size of 1,000,000 took an incredible 66,763 ms seconds with Firestore while MySQL required only 905 ms. In this query scenario, a multiple linear regression analysis shows Firestore's latency was 12x more affected by a rise in r than n.

MySQL is the clear winner in raw latency performance, especially once complex combinations of conditions are used while querying. Yet, for a one-month billing cycle involving writing, storing and reading 1,000,000 charger structures, Firestore costs were $1.17 while MySQL was $35.03. The MySQL server was perpetually deployed and incurring costs regardless of whether there was any user traffic. With the energy auction in its early development stages, the cost difference is a deciding factor. Also, the flexibility of the schemaless document-model allows features to be deployed in stages with the persisted data adapting to match. Finally, as the auction grows (we hope!), the serverless document-model seamlessly scales horizontally without limit to accommodate both capacity increase and geographical diversity although it may eventually cost more with very high transaction volumes. In conclusion, the NoSQL, non-relational, serverless database is the best choice. SQL may be the premium solution of the past but NoSQL and Firestore are the way of the future.

References

[1] Airbnb, 2022. URL https://www.airbnb.com. Accessed: 2022-08-13.

[2] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. ISSN 0163-5808. doi: 10.1145/1978915.1978919. URL https://doi.org/10.1145/1978915.1978919.

[3] D. D. Chamberlin. Early history of SQL. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012. doi: 10.1109/MAHC.2012.61.

[4] D. Hand et al. Statistical challenges of administrative and transaction data. *Journal of the Royal Statistical Society. Series A (Statistics in Society)*, 181(3):pp. 555–605, 2018. ISSN 09641998, 1467985X. URL https://www.jstor.org/stable/48547504.

[5] X. Dong and X. Li. A novel distributed database solution based on MySQL. In *2015 7th International Conference on Information Technology in Medicine and Education (ITME)*, pages 329–333, 2015. doi: 10.1109/ITME.2015.48.

[6] EVAdoption. US charging network rankings, 2022. URL https://evadoption.com/ev-charging-stations-statistics/us-charging-network-rankings.

[7] G. Brito et al. Migrating to GraphQL: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–150, 2019. doi: 10.1109/SANER.2019.8667986.

[8] Google. Firestore, 2023. URL https://cloud.google.com/sql/firestore. Accessed: 2023-06-19.

[9] Google. Pricing: Firestore, 2023. URL https://cloud.google.com/firestore/pricing. Accessed: 2023-06-22.

[10] Google. Cloud SQL for MySQL, 2023. URL https://cloud.google.com/sql/mysql. Accessed: 2023-06-19.

[11] Google. Pricing: Cloud SQL MySQL, 2023. URL https://cloud.google.com/sql/pricing#mysql-pg-pricing. Accessed: 2023-06-22.

[12] GraphQL Foundation. A query language for your API, 2023. URL https://graphql.org. Accessed: 2023-06-23.

[13] H. Khazaei et al. How do i choose the right NoSQL solution? a comprehensive theoretical and experimental survey. *Big Data and Information Analytics*, 1(2):185–216, 2016.

[14] R. Irle. The electric vehicle world sales database, 2023. URL https://www.ev-volumes.com. Accessed: 2023-06-16.

[15] J. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug 2013. ISSN 0734-2071. doi: 10.1145/2491245. URL https://doi.org/10.1145/2491245.

[16] J. Moon et al. A heterogeneous IoT data analysis framework with collaboration of edge-cloud computing: Focusing on indoor pm10 and pm2. 5 status prediction. *Sensors*, 19(14):3038, 2019.

[17] M. Eyada et al. Performance evaluation of IoT data management using MongoDB versus MySQL databases in different cloud environments. *IEEE Access*, 8:110656–110668, 2020. doi: 10.1109/ACCESS.2020.3002164.

[18] Microsoft. Analyze data in Excel, 2023. URL https://support.microsoft.com/en-us/office/analyze-data-in-excel-3223aab8-f543-4fda-85ed-76bb0295ffc4. Accessed: 2023-06-21.

[19] S. Minos. New plug-in electric vehicle sales in the United States nearly doubled from 2020 to 2021. Technical report, US Department of Energy, March 2022. URL https://www.energy.gov/energysaver/articles/new-plug-electric-vehicle-sales-united-states-nearly-doubled-2020-2021.

[20] N. Jatana et al. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[21] D. Ongaro and J. Ousterhout. The Raft consensus algorithm, 2023. URL https://raft.github.io/. Accessed: 2023-07-04.

[22] R. Pandey. Performance benchmarking and comparison of cloud-based databases MongoDB (NoSQL) vs MySQL (relational) using YCSB. Technical report, Technical Report. https://doi. org/10.13140/RG. 2.2. 10789.32484, 2020.

[23] J. Pokorny. NoSQL databases: A step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS '11, page 278–283, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307840. doi: 10.1145/2095536.2095583. URL https://doi.org/10.1145/2095536.2095583.

[24] R. Bannon et al. Mysql conceptual architecture. *Technical report, University of Waterloo*, 2002.

[25] R. Kesavan et al. Firestore: The NoSQL serverless database for the application developer. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3367–3379, 2023.

[26] Solar Energy Industries Association. Solar data facts sheet, 2022. URL https://www.seia.org/research-resources/solar-data-cheat-sheet. Accessed: 2022-08-13.

[27] Stripe. Stripe payment processing platform for the Internet, 2023. URL https://www.stripe.com. Accessed: 2023-06-16.

[28] T. Taipalus and V. Seppänen. SQL education: A systematic mapping study and future research agenda. *ACM Trans. Comput. Educ.*, 20(3), Aug 2020. doi: 10.1145/3398377. URL https://doi.org/10.1145/3398377.

[29] Uber, 2022. URL https://www.uber.com. Accessed: 2022-08-13.

[30] U.S. Department of Energy. Alternate fueling station locator, 2022. URL https://afdc.energy.gov/stations/#/analyze. Accessed: 2022-08-02.

[31] R. Yadav and A. Rahut. FlexiRaft: Flexible Quorums with Raft. *The Conference on Innovative Data Systems Research (CIDR)*, 2023.

[32] Yahoo! Yahoo cloud serving benchmark, 2010. URL https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/. Accessed: 2023-06-19.