Unlock the FoundationDB Record Layer: Language Flexibility and Network Freedom with a gRPC Microservice

James Krepelka
jkrepelka@csuchico.edu
California State University Chico
Chico, California, USA

ABSTRACT

This paper examines the FoundationDB Record Layer, a Java library enabling record-oriented data storage within the scalable NewSQL FoundationDB database. While powerful, the Record Layer's Java-only implementation and network co-location requirement limit its broader adoption. This research explores the feasibility of encapsulating the Record Layer within a gRPC microservice to address these limitations.

A gRPC microservice layer was implemented, and its performance impact was evaluated against direct Record Layer usage. Experimental results demonstrate that while some overhead is introduced, particularly for read operations, the gRPC approach significantly expands language accessibility and architectural flexibility. For write-heavy workloads, the performance overhead is minimal (1-6%). This makes the gRPC-enabled Record Layer a compelling option for developers seeking FoundationDB's scalability and ACID guarantees without language or network constraints.

Future work directions include optimizing protobuf handling within the microservice and exploring the development of a "Relationship Layer" to support richer relational data modeling on top of the Record Layer.

CCS CONCEPTS

• **Information systems** → **Database web servers**; *Key-value stores.*

KEYWORDS

FoundationDB, distributed databases, gRPC

ACM Reference Format:
James Krepelka. 2024. Unlock the FoundationDB Record Layer: Language Flexibility and Network Freedom with a gRPC Microservice. In *Proceedings of California State University Chico (CSU Chico).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Database Evolution and the Rise of NewSQL

Relational databases built upon SQL have dominated data storage since the 1970s [6]. However, the scalability demands of Web 2.0 applications exposed limitations in traditional SQL systems. This led to the emergence of NoSQL databases that prioritized scalability and availability, often at the cost of strict consistency guarantees [1]. While offering advantages, the lack of ACID transactions in many NoSQL systems created complexity for developers accustomed to the transactional guarantees of SQL [1].

NewSQL technologies, such as FoundationDB, emerged to bridge this gap. They seek to provide scalable, distributed data storage while maintaining ACID compliance. FoundationDB takes this concept further by proposing a fundamental separation of data models, query languages, and storage engines[13]. This approach emphasizes the key-value store as a basic building block, with higher-order data models and query capabilities implemented as composable "Layers."

1.2 FoundationDB's Record Layer

FoundationDB's Record Layer (2018) is a notable example of this layered approach. It provides a record-oriented storage model akin to relational databases, featuring structured types, indexes, and basic querying capabilities. This layer is of particular interest because it illustrates both the possibilities and potential limitations of FoundationDB's design philosophy.

The Record Layer is primarily vended as a Java library and interacts with each node in the FoundationDB cluster as needed. This means that the primary use case supported by the Record Layer is to be imported and used by a Java application that runs in the same network as the FoundationDB cluster itself.

1.3 Research Focus and Thesis

This paper examines the FoundationDB Record Layer in the context of contemporary database trends. I suggest that the Record Layer's Java-only implementation limits its potential impact and broader adoption, and the network co-location requirement hinders modern deployment practices and security models. Exposing the FoundationDB Record Layer through a microservice would significantly expand its accessibility and compatibility with modern development practices, so long as it doesn't compromise the performance of interactions with FoundationDB.

2 LITERATURE REVIEW

2.1 SQL and the Origins of Relational Databases

Modern databases have their roots in Dr. E.F. Codd's 1970 proposal of the relational model [6] led to the introduction of Structured English Query Language (SEQUEL, later shortened to SQL) [4]. SQL technologies dominated the data storage landscape until the rise of Web 2.0 and its unprecedented scaling demands.

2.2 The Rise of NoSQL and the CAP Theorem

Web 2.0 companies faced limitations when trying to scale traditional SQL databases. This led to the emergence of NoSQL technologies

CSU Chico, May 14, 2024, Chico, CA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

in the mid-2000s, such as Google's Bigtable (2005)[5], Amazon Dynamo (2007)[7], and Facebook's Cassandra (2009)[9]. These solutions were designed for high scalability in the face of massive data growth.

However, Eric Brewer, in his 2000 keynote, presented the CAP theorem [2, 3]. This theorem posits that distributed systems can only guarantee two out of three properties: Consistency, Availability, and Partition tolerance [8]. NoSQL's emphasis on availability and partition tolerance often came at the expense of consistency, creating challenges for developers in ensuring data integrity.

2.3 ACID and the Emergence of NewSQL

The lack of ACID (Atomicity, Consistency, Isolation, Durability) compliance in NoSQL systems became a barrier for many developers [1]. NewSQL technologies emerged in the early 2010s, aiming to bridge the gap by offering ACID compliance alongside horizontal scalability [1]. FoundationDB, released in 2010 [11], gained prominence for its highly scalable, fault-tolerant design and flexible data models [10, 13]. FoundationDB's emphasis on ACID transactions enabled developers to regain the transactional guarantees familiar from traditional SQL systems.

2.4 FoundationDB's Innovations and Differentiators

FoundationDB stands out with its core design philosophy of separating the data model, query language, and storage engine [13]. This allows developers to build custom "Layers" to store and access data in diverse formats, including tables, documents, and graphs. FoundationDB's Record and Document Layers demonstrate this flexibility. The Record Layer provides features familiar to SQL users, while the Document Layer acts as a drop-in replacement for MongoDB, easing migration [12].

2.5 FoundationDB's Shortcomings and Potential

Despite its advantages, FoundationDB faces limitations that may hinder its wider adoption. The Record Layer's restriction to Java and its requirement to be on the same network as the cluster are potential barriers. Addressing these limitations could significantly expand FoundationDB's reach.

3 EVALUATION ENVIRONMENT SETUP

This section provides a comprehensive overview of all the components of the experiment (Figure 1). I will describe the process of writing the code for the experiment and the configuration of the testing environment in detail.

3.1 Code

To evaluate the time added by a gRPC API built around the Record Layer, I built a common TicketLayer library that the two different CLIs use.

3.1.1 TicketLayer Library. I implemented a shared TicketLayer library as a Data Access Layer (DAL). This DAL provides consistent save and get operations for interacting with the FoundationDB Record Layer. Employing this shared DAL across both the CLI and server components guarantees experimental consistency.

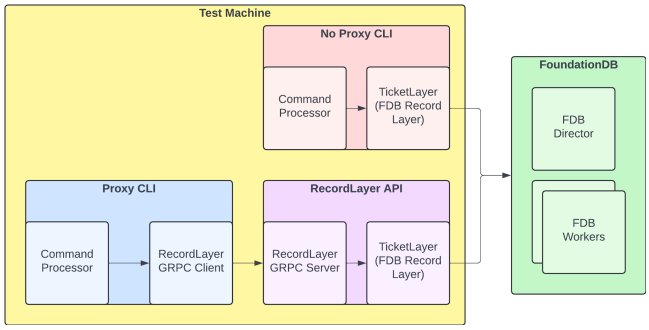

Figure 1: An overview of the system architecture for the experiment.

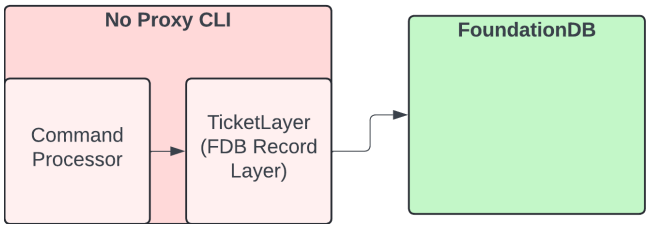

Figure 2: An overview of how the No Proxy CLI directly interacts with FoundationDB.

3.1.2 No Proxy CLI. The No Proxy Command-Line Interface (CLI) is implemented in the Java programming language. It leverages the Record Layer as a third-party library and establishes network connections directly with FoundationDB nodes. The CLI comprises two distinct components: a command processor that accepts user commands for interacting with FoundationDB, and the TicketLayer library, which utilizes the FoundationDB Record Layer library to facilitate connectivity (Figure 2).

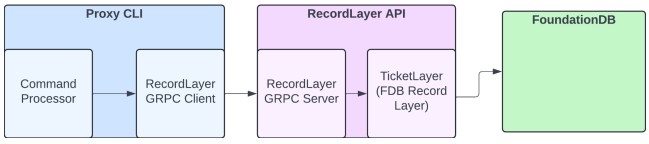

Figure 3: An overview of how the Proxy CLI interacts with the RecordLayer API, which then interacts with FoundationDB.

3.1.3 Proxy CLI. The Proxy CLI approach involves two Java binaries: the Proxy CLI and the RecordLayer API server (Figure 3).

This Java implementation maintains consistency with the No Proxy CLI for comparative testing. Importantly, the Proxy CLI could be written in any language supported by gRPC (currently 11, see https://grpc.io/docs/languages/).

The Proxy CLI includes a command processor, mirroring the No Proxy CLI for consistent user interaction, and a generated gRPC client library. The RecordLayer API server comprises a generated gRPC server and the shared TicketLayer library for FoundationDB access.

This scenario evaluates the performance overhead introduced by the gRPC layer in facilitating FoundationDB interactions.

3.2 Hardware Configuration

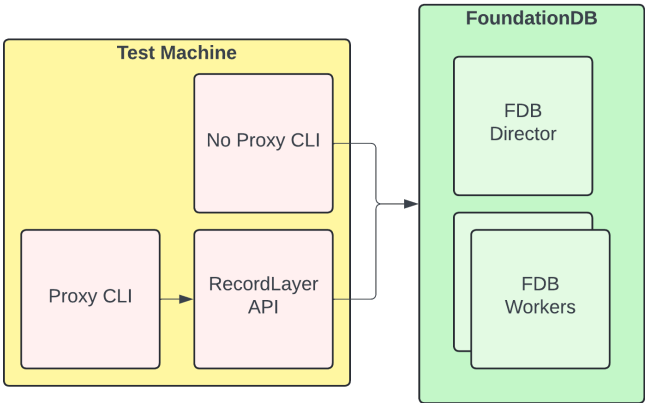

Figure 4: An overview of the machine configuration and locations where the CLIs were executed.

For the purpose of acquiring timing-related data for this experiment, four individual machines were utilized (Figure 4), all of which were operational within the confines of a singular server rack.

Since the focus is on the overhead of the gRPC layer, detailed hardware specifications of the FoundationDB cluster are less critical. However, providing a high-level overview ensures reproducibility.

3.2.1 Test Machine.

- `OS`: Linux 6.8.2
- `RAM`: 256 GiB (3200MHz DDR4)
- `Storage`: 20+ TB NVMe (ZFS)
- `Network`: 1GB link to FoundationDB cluster (same rack)

3.2.2 FoundationDB Cluster (3 nodes).

- `foundationdb.conf`: paper/supporting/foundationdb.conf
- `OS`: Linux 6.8.2
- `RAM`: 128 GiB per node (3200MHz+, one at 4800MHz)
- `Storage`: 20+ TB NVMe (ZFS)
  - `Log`: 54GiB Intel® Optane™ P1600X
  - `Data`: 2TB Crucial MX500 or 2TB 870 EVO SATA SSDs
- `Network`: 10GB interconnects

4 EVALUATION DATA

4.1 Experimental Methodology

Four performance tests were conducted, each comparing the Proxy CLI and No Proxy CLI setups described in Section 2. Both CLIs executed the same commands with incrementally increasing arguments. To ensure accurate timing, a timing mechanism was implemented around each command. This process was repeated 20 times per CLI and argument combination. Data points in subsequent charts represent the average runtime across these 20 executions, with a measured confidence level exceeding 99.9

4.2 Single Read Evaluation

To assess single record retrieval, the **get-single N** command was executed on both Proxy and No Proxy CLIs with increasing values of **N** (1,000 to 40,000). The test dataset consisted of 75,000 randomly generated 'Ticket' objects stored in FoundationDB. Prior to timed runs, 1,000 test objects were randomly selected from the dataset. Performance measurements were taken as each CLI retrieved these 1,000 objects individually.

4.2.1 Data. Below is the collected data in bar-chart and tabular formats.

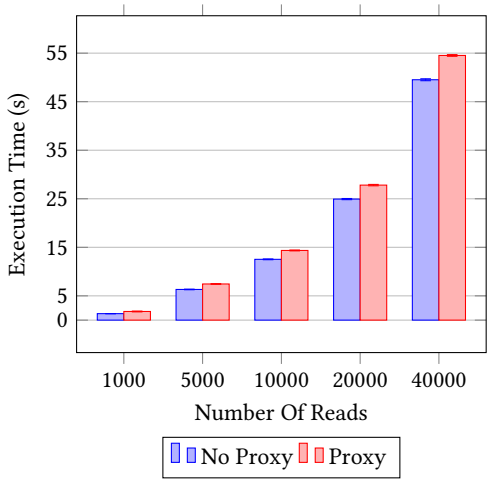

Figure 5: The number of reads plotted against the time it took to execute them.

N	NoProxy (ms)	Proxy (ms)	% Inc
1,000	1,335.40 +/- 15.48	1,782.35 +/- 80.91	33.47%
5,000	6,319.50 +/- 60.42	7,447.65 +/- 72.04	17.85%
10,000	12,525.05 +/- 81.47	14,354.10 +/- 87.04	14.60%
20,000	24,936.70 +/- 119.98	27,815.0 +/- 143.04	11.54%
40,000	49,522.80 +/- 199.89	54,526.0 +/- 173.46	10.10%

Table 1: The number of reads, how long they took to execute in milliseconds, +/- the measured error, and the percentage increase in runtime of the Proxy CLI versus the NoProxy CLI.

4.2.2 Partial Conclusion. The data demonstrates that a gRPC API layer built upon the Record Layer introduces considerable overhead when retrieving single records from the database. In scenarios with a low volume of reads, both runtime increases (33%) and greater variability (80ms vs 15ms) were observed compared to direct Record Layer access. While the overhead decreases with larger numbers of reads (10%), it remains noticeable.

This finding suggests that for applications requiring high Queries Per Second (QPS) on small-object reads, the performance trade-offs of a gRPC layer warrant careful consideration against the architectural and developer-facing benefits outlined earlier in the paper.

4.3 Batch Read Evaluation

To evaluate batch read performance, the **get-multiple N** command was executed on both Proxy and No Proxy CLIs with increasing values of **N** (10 to 400). The test involved retrieving the entire dataset of 75,000 randomly generated 'Ticket' objects from FoundationDB. Each retrieval operation was repeated **N** times within a timed execution.

4.3.1 Data. Below is the collected data in bar-chart and tabular formats.

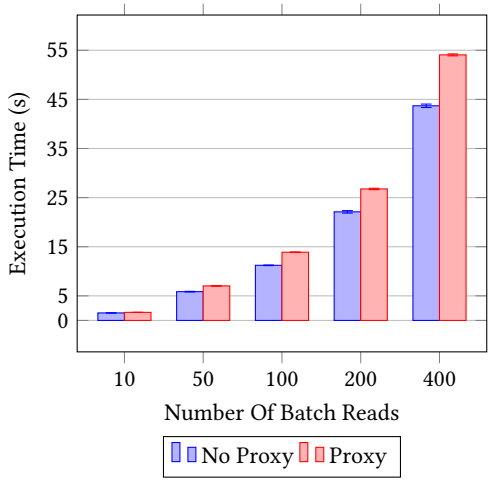

Figure 6: The number of batch reads plotted against the time it took to execute them.

N	NoProxy (ms)	Proxy (ms)	% Inc
10	1,526.35 +/- 15.84	1,642.05 +/- 18.47	7.58%
50	5,855.75 +/- 66.25	7,023.70 +/- 51.00	19.95%
100	11,220.40 +/- 63.00	13,883.55 +/- 54.98	23.73%
200	22,105.75 +/- 250.38	26,766.95 +/- 142.92	21.09%
400	43,695.85 +/- 348.24	54,060.25 +/- 199.79	23.72%

Table 2: The number of batch reads, how long they took to execute in milliseconds, +/- the measured error, and the percentage increase in runtime of the Proxy CLI versus the NoProxy CLI.

4.3.2 Partial Conclusion. Experimental data indicates that the gRPC API layer introduces a non-negligible overhead for batch reads from the database. For smaller numbers of repeated batch reads, a runtime increase of 7.58% was observed compared to direct Record Layer access. This overhead grows to approximately 20%, with increased variability, as the number of repeated batch reads scales.

While less pronounced than in single-read scenarios, these results suggest that the architectural benefits of a gRPC layer should be weighed against its performance impact, particularly in applications with frequent, large-scale read operations.

4.4 Single Write Evaluation

To assess single-object write performance, **the create-single N** command was executed on both Proxy and No Proxy CLIs with increasing values of **N** (1,000 to 40,000). For each test iteration, **N** randomly generated 'Ticket' objects were created. The timer was initiated prior to saving these objects individually to FoundationDB. To maintain a consistent database state, all created objects were deleted before subsequent test runs.

4.4.1 Data. Below is the collected data in bar-chart and tabular formats.

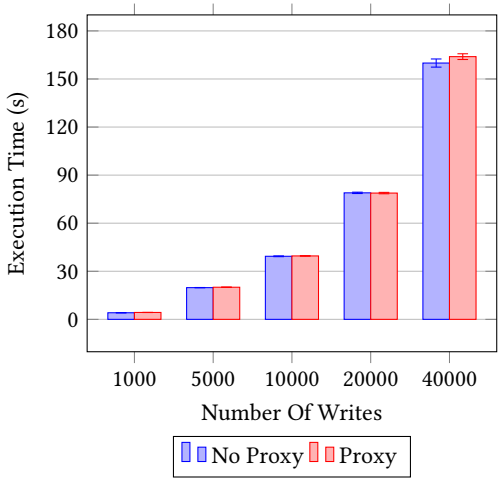

Figure 7: The number of writes plotted against the time it took to execute them.

N	NoProxy (ms)	Proxy (ms)	% Inc
1,000	4,113.50 +/- 70.66	4,305.95 +/- 49.32	4.68%
5,000	19,791.80 +/- 90.93	20,045.70 +/- 126.23	1.28%
10,000	39,358.35 +/- 238.27	39,577.40 +/- 185.76	0.56%
20,000	78,946.55 +/- 420.61	78,803.85 +/- 433.98	-0.18%
40,000	159,947.95 +/- 2,544.05	163,947.50 +/- 1,787.43	2.50%

Table 3: The number of writes, how long they took to execute in milliseconds, +/- the measured error, and the percentage increase in runtime of the Proxy CLI versus the NoProxy CLI.

4.4.2 Partial Conclusion. The experimental results indicate that a gRPC API layer introduces minimal overhead when writing small records to the database. In tests with a low volume of writes, a slight runtime increase (4.68%) was observed for the Proxy CLI. However, this difference becomes negligible as the number of writes increases.

These findings suggest that for write-heavy workloads, employing a gRPC proxy layer has limited performance drawbacks. Consider this alongside the architectural and developer-facing benefits discussed earlier when choosing a FoundationDB Record Layer access strategy

4.5 Batch Write Evaluation

For the purpose of conducting this experiment, I executed the **create-multiple 500 N** command for both the Proxy and No Proxy CLIs, gradually increasing the number of tickets from 10 to 400. During this test, each CLI generated **500** random ticket objects **N** times, storing them in distinct lists of **500** tickets each. Subsequently, the CLIs initiated a timer and proceeded to access each list in FoundationDB via **N** distinct calls.

Upon completion of the test, I proceeded to delete all tickets from FoundationDB to ensure that subsequent creations occurred within a pristine and empty database.

4.5.1 Data. Below is the collected data in bar-chart and tabular formats.

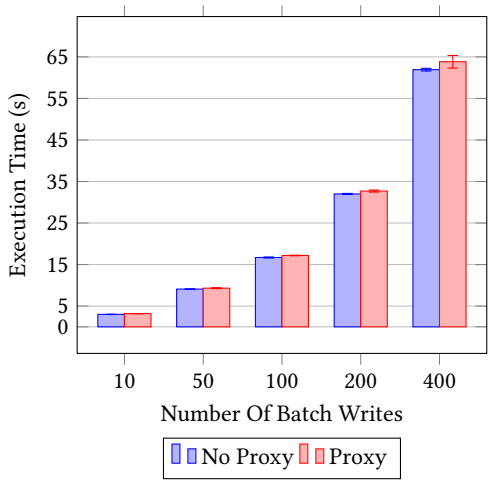

Figure 8: The number of batch writes plotted against the time it took to execute them.

N	NoProxy (ms)	Proxy (ms)	% Inc
10	3,005.15 +/- 42.41	3,185.90 +/- 29.57	6.0%
50	9,091.70 +/- 70.42	9,317.05 +/- 114.41	2.5%
100	16,699.30 +/- 122.69	17,182.35 +/- 73.59	2.9%
200	32,001.25 +/- 132.21	32,699.90 +/- 248.54	2.2%
400	61,929.30 +/- 327.98	63,828.75 +/- 1,523.45	3.1%

Table 4: The number of batch writes, how long they took to execute in milliseconds, +/- the measured error, and the percentage increase in runtime of the Proxy CLI versus the NoProxy CLI.

4.5.2 Partial Conclusion. From the aforementioned data, it can be observed that the inclusion of a gRPC API layer on top of the Record Layer does not result in a significant increase in the overhead associated with writing large chunks of records to a backend database. In the case of small numbers of repeated batch writes, a slightly increased runtime (6%) is observed when utilizing the Proxy CLI as opposed to the CLI. However, as the number of writes increases, there appears to be negligible difference between the gRPC proxy and non-gRPC versions.

Given this data, it is reasonable to assert that employing the gRPC proxy version for storing large objects in FoundationDB is a prudent choice, with the exception of systems where performance is of paramount importance.

5 CONCLUSION AND FUTURE WORKS

5.1 Overall Conclusion

The experimental results strongly support the thesis that encapsulating the FoundationDB Record Layer within a gRPC microservice offers significant advantages in developer accessibility and architectural flexibility, while introducing acceptable performance overheads.

While a multi-language rewrite of the Record Layer would provide the most optimized solution, the development and maintenance costs of such an approach likely outweigh the benefits for most use cases. The gRPC-based microservice approach effectively addresses the language barrier with minimal performance impact, particularly for write-heavy workloads where latency increases are within a 1-6

Crucially, deploying the Record Layer as a horizontally scalable microservice resolves the network co-location requirement by handling the direct connection to FoundationDB for each request. This empowers organizations to leverage FoundationDB's open-source, NewSQL capabilities within modern, security-conscious network architectures. Overall, this work demonstrates that a gRPC microservice wrapper effectively expands the potential of the FoundationDB Record Layer, making it a more compelling data storage option for a wider range of developers and applications.

5.2 Future Work

5.2.1 Remove "Business Logic" From The API. A key optimization opportunity lies in streamlining the protobuf handling within the gRPC microservice and Record Layer. Currently, protobuf messages undergo unnecessary decoding and re-encoding steps between the microservice and storage layers. By modifying the Record Layer to directly accept and store protobuf messages received from gRPC clients, performance improvements and easier extensibility could be realized.

5.2.2 Create a Relationship Layer. This research highlights the potential benefits of extending the layered approach within FoundationDB. Informed by the limitations of the Record Layer in handling complex relational data, a valuable future direction lies in developing a "Relationship Layer" on top of the existing Record Layer and gRPC API. This layer would introduce support for defining and querying rich relationships between records.

By leveraging the core infrastructure and this gRPC microservice, developers could enjoy familiar SQL-like workflows while utilizing FoundationDB's scalability and resilience. Offering both the Record Layer and Relationship Layer would empower developers to choose the optimal data modeling approach based on their application's specific needs.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to a good friend of mine who not only directed my attention towards FoundationDB, but also provided the evaluation environment described above. Scott Minor supported me with supporting knowledge, advice, and willpower as I needed them. I wouldn't have looked into this without him, nor would any paper I eventually wrote have ended up half as good. Thank you Scott!

I'd also like to thank my Graduate Project Advisor for this, David Zeichick at CSU Chico. He helped guide me through the structure of researching a project and his weekly check-ins kept me on track to get all of this done in a single semester.

REFERENCES

[1] Matt Aslett. 2011. How will the database incumbents respond to NoSQL and NewSQL? https://cs.brown.edu/courses/cs227/archives/2012/papers/newsql/aslett-newsql.pdf

[2] Eric A. Brewer. 2000. Towards Robust Distributed Systems. https://www.podc.org/podc2000/brewer.html

[3] Julian Browne. 2009. Brewer's Cap Theorem. https://www.julianbrowne.com/article/brewers-cap-theorem/

[4] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language *(SIGFIDET '74)*. Association for Computing Machinery, New York, NY, USA, 249–264. https://doi.org/10.1145/800296.811515

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Seattle, WA, USA, 205–218.

[6] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (jun 1970), 377–387. https://doi.org/10.1145/362384.362685

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[8] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (Jun 2002), 51–59. https://doi.org/10.1145/564585.564601

[9] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a P2P network. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (Calgary, AB, Canada) *(PODC '09)*. Association for Computing Machinery, New York, NY, USA, 5. https://doi.org/10.1145/1582716.1582722

[10] Ashish Motivala. 2018. How foundationdb powers snowflake metadata forward. https://www.snowflake.com/blog/how-foundationdb-powers-snowflake-metadata-forward/

[11] FoundationDB Team. 2018. https://www.foundationdb.org/blog/foundationdb-is-open-source/

[12] FoundationDB Team. 2021. https://github.com/FoundationDB/fdb-document-layer

[13] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2653–2666. https://doi.org/10.1145/3448016.3457559