Assembling a Query Engine From Spare Parts

Mosha Pasumansky
Firebolt Analytics
moshap@firebolt.io

Benjamin Wagner
Firebolt Analytics
benjamin.wagner@firebolt.io

ABSTRACT

Building a new cloud data warehouse is a daunting challenge, requiring massive investments into both the query engine and surrounding cloud infrastructure. Given the mature space, it might seem like a Herculean task to enter the market as a small startup.

At Firebolt we assembled a working, high-performance cloud data warehouse in less than 18 months. We achieved this by building our query engine on top of existing projects and then investing heavily into differentiating features. This paper presents our decision-making and learned lessons along the way.

1 INTRODUCTION

Firebolt is a modern cloud data warehouse built to support user-facing, data-intensive applications [12]. These workloads are challenging, as users expect queries to return in tens of milliseconds. Additionally, user-facing applications can have thousands of simultaneous users and queries. They exhibit many queries-per-second (QPS), as well as high concurrency.

Building a database management system is hard, as it consists of multiple complex components. These include a query engine, storage engine, transaction manager, and a system catalog. This challenge is amplified when building a cloud data warehouse, as it adds additional layers of complexity to the system. This includes cloud platform infrastructure, cloud storage management, SaaS components, and much more. This is exemplified in Figure 1, presenting Firebolt's high-level architecture.

All of these components are necessary even for a barebones system. On top of that, significant engineering effort needs to be invested to build out differentiating features.

Even for a large team, building such a system from scratch would take multiple years. At Firebolt, we were able to launch our cloud data warehouse for real customers running production workloads in under 18 months. In order to achieve this, we assembled Firebolt from multiple existing components. While some only required small modifications, others were used as a starting point and ended up changing significantly.

This paper presents how we assembled our high-performance query engine from existing components. It is structured as follows. Section 2 focuses on the decisions made while building out the engine itself. Afterwards, Section 3 describes how we leverage open-source tools in order to continuously test our query engine. We summarize our lessons learned in Section 4 and conclude in Section 5.

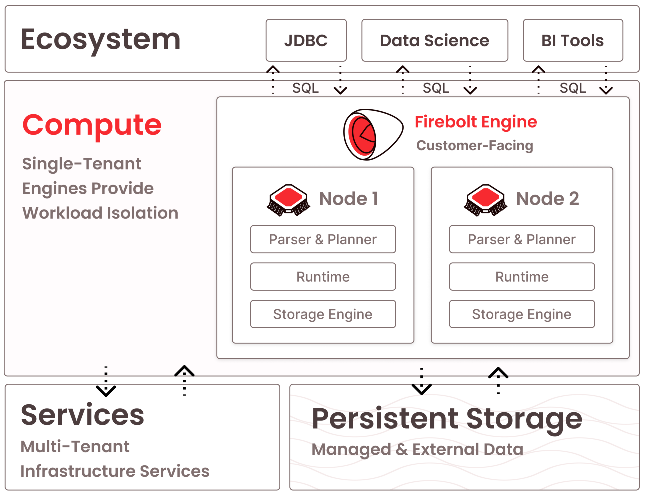

Figure 1: The high-level architecture of the Firebolt cloud data warehouse. Firebolt revolves around the concept of engines. An engine in Firebolt provides a set of isolated compute resources. The nodes of an engine run the parser, planner, runtime, and storage engine. Our ecosystem integrations communicate with engines through SQL. Surrounding services provide essential functionality such as infrastructure provisioning and metadata management.

2 QUERY ENGINE

This Section outlines the open-source building blocks used to assemble our query engine. The design of Firebolt's engine follows a textbook separation of components [21].

Our SQL parser accepts Firebolt's SQL dialect and converts the user query into an abstract syntax tree (AST). The logical planner then transforms this AST into a logical query plan (LQP). To achieve this, it uses logical metadata such as table and view definitions, data types, and the function catalog. The planner then applies logical transformations to produce an optimized LQP. Sections 2.1 and 2.2 present how we chose an existing project as a foundation for these components at Firebolt.

Afterwards, Firebolt's physical planner constructs a distributed query plan (DQP) from the LQP. To achieve this, it uses physical metadata such as the existence of indexes, table cardinalities, and data distribution. Our distributed runtime orchestrates the execution of the DQP across a cluster of Firebolt nodes. The responsibilities include scheduling, data exchange between stages, and query fault tolerance. Finally, the local runtime executes relational operators within a single Firebolt node. Section 2.3 outlines how we decided on an existing project to lay the foundations for our runtime.

We also present some of the engineering challenges encountered when basing the planner and runtime on existing projects. Section 2.4 shows how communication between the planner and runtime evolved over time. Our push towards a custom-built distributed runtime is outlined in Section 2.5.

2.1 SQL Dialect

Cloud data warehouses do not exist in a vacuum – they are part of a wider data ecosystem. This ecosystem encompasses tools for ETL/ELT, BI, reporting, data science, ML and data observability. Examples include Fivetran, Dbt, Tableau, Looker and Monte-Carlo. This is outlined in Figure 2. As users build their applications on top of these tools, having extensive integrations with the wider ecosystem is critical to Firebolt's success. After all, nobody wants a cloud data warehouse with no tools that can talk to it!

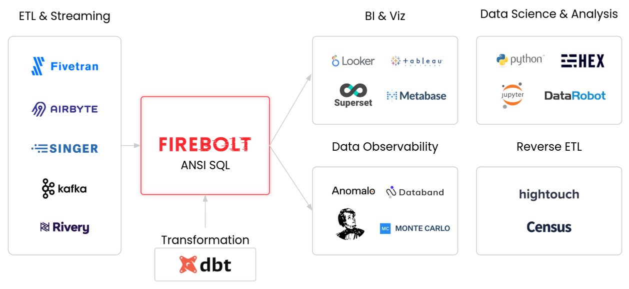

Figure 2: As a cloud data warehouse, Firebolt has to integrate with the wider ecosystem.

Fortunately, all of the above tools talk to the data warehouse using SQL. This greatly simplifies integrating with the ecosystem. Challenges remain nevertheless. Despite the existence of the ANSI SQL standard, virtually every database features its own SQL dialect. As a result, the above tools require a variety of custom drivers, connectors and adaptors to support different database systems.

To be a successful startup within the cloud data warehousing space and to satisfy customers, integrating with the ecosystem from day one is essential. As a small startup, Firebolt cannot expect large companies behind ecosystem tools to spend time and resources dedicated to adding support through custom connectors and drivers.

To ease ecosystem integration, we decided that the Firebolt SQL dialect should be similar to an existing, widely adopted SQL dialect. Choosing the Postgres dialect as the north star was an easy choice. It is hugely popular and highly compliant with standard ANSI SQL. Almost every tool in the data stack supports Postgres SQL as a result.

It is important to note that being compatible with Postgres SQL does not necessitate being compatible with the Postgres wire protocol [19]. Our drivers communicate with Firebolt through a custom HTTP-based REST protocol.

2.2 SQL Parser and Planner

As outlined in the previous section, it was a requirement for Firebolt's SQL parser to be very similar to Postgres SQL. It has to fully cover DDL, DML, and DCL statements. However, support for DQL (i.e. SELECT) statements is the most important, as they make up the bulk of the workload.

We wanted to base the logical planner on an existing project meeting a broad set of requirements. The planner needed to support the most important rules in modern data warehousing, such as predicate pushdown and subquery decorrelation. As part of this, the project needed to also have an extensible framework for rule-based transformations. This would allow us to easily add Firebolt specific rules as we scaled the product.

In addition to rule-based transformations, the planner needed to support cost-based join reordering. This includes allowing us to build custom statistics sources and cost models. This is especially important given the variety of different index types supported by Firebolt. For example, we use sparse primary and secondary indexes for data pruning, as well as specialized indexes for frequently occurring broadcast joins [24].

Finally, the planner needed to support composite data types such as arrays and row (struct) types. These are popular for the user-facing, data-intensive applications targeted by Firebolt.

It is possible to pick different projects as the baseline for the parser and planner, respectively. However, these two components share a very complex interface: the AST of a query leaving the parser and entering the planner for semantic analysis. We decided to give preference to projects that include both a parser and a planner.

Luckily, there are a variety of open-source projects that we were able to consider as a basis for Firebolt's SQL parser and planner. These are outlined in the following.

Postgres Parser. Using the Postgres parser directly would have been the obvious choice given our desire to be compliant with Postgres SQL. This approach has been used successfully in multiple other systems [8, 20]. Isolating the parser from the Postgres code has been done by the libpg_query project[1]. It packages the original C-based Postgres parser in a library [13].

While this makes it reasonably easy to build a system on top of the Postgres parser, it is difficult to isolate the planner code without bringing in the rest of the large Postgres codebase. For us, this meant that while going with Postgres would require almost no investment in the parsing layer, it would require significant effort to build a production-grade planner that meets our needs.

ZetaSQL. ZetaSQL is a parser and analyzer from Google[2] built in C++. It is an open-source port of GoogleSQL [23]. GoogleSQL powers the cloud products BigQuery [17], Spanner [3] and Dataflow, as well as the Google internal products Dremel, F1 [22], and Procella [7]. It is a cleanly built, extensively tested, and production-ready system.

However, ZetaSQL provides an opinionated dialect that disagrees with Postgres SQL in many basic features. Additionally, ZetaSQL only supports rudimentary transformations and no feature-rich planner.

Calcite. Apache Calcite is a framework that provides query processing, optimization, and query language support for data processing

[1]https://github.com/pganalyze/libpg_query
[2]https://github.com/google/zetasql

systems [4]. It includes parsers for multiple SQL dialects and a modular and extensible query planner with support for pluggable rules. Calcite is well-built and battle tested. It is used in many high-profile open-source systems such as Apache Hive, Apache Storm, Apache Flink, Druid, and MapD[3].

Compared to the other alternatives we considered that are written in C++, Calcite is implemented in Java.

DuckDB. DuckDB is an in-memory, in-process analytical database system originating from CWI [20]. It is extensively tested and widely used for interactive data analysis. DuckDB's query planner supports both rule-based optimizations and cost-based join reordering. DuckDB uses the libpg_query project as a baseline for their parser, providing Postgres SQL complicance. Nowadays, DuckDB ported its parser over to C++ [4].

At the time we decided on a project to base our parser and planner on, DuckDB was significantly less mature than it is today.

Hyrise. Hyrise is an in-memory database developed at HPI [11]. It has a relatively simple code base, making it easy to refactor and extend. Similar to DuckDB, it supports rule-based optimizations and cost-based join reordering.

However, as an academic project, Hyrise is not battle tested and does not have extensive SQL coverage.

We decided early on that we wanted planner and runtime to be written in the same programming language. We do not think that this is a hard requirement when building a new system. There are multiple successful systems with JVM-based planners and high-performance runtimes written in C++ [5, 10]. Nevertheless, we believe that the engine being written in one language allows us to have high velocity as a startup. A lot of work in database systems happens at the intersection of planner and runtime. This is especially true for Firebolt, as we had to integrate both components originating from different open-source projects. Planner and runtime being written in the same programming language makes it easy for developers to work across the stack with minimal context switching. As we decided on a runtime written in C++, we wanted the planner to follow suit. This left us with a choice of either Hyrise or DuckDB.

We decided to build upon the Hyrise project due to its simplicity and extensibility. While getting Firebolt production ready, the fact that Hyrise is an academic system with limited SQL support proved to be a challenge. For engineers building a new database system today, using DuckDB is most likely a better starting point. This is because DuckDB has matured significantly since we started building Firebolt and is widely used nowadays.

At the same time, using Hyrise did prove to be a good choice for our use case. Indeed, our initial instinct to go for it given its relative simplicity turned out to be true. We invested heavily to make Firebolt's parsing and planning layer production ready. Nowadays, its open-source roots in Hyrise are hard to recognize. Our additions include wide ranging extensions to the parser to give good SQL coverage and considerable changes to the planning layer. Among other things, we have added extensive support for composite data types, changed the representation of the logical query plan to make it easier to build rule-based optimizations and added a variety of new rules to the planner.

Our redesign of the logical query plan was heavily inspired by Calcite. As it is the gold standard of extensible and feature-rich open-source query planners, we decided to utilize many of the concepts found in Calcite's relational algebra representation. This allows our planner to be future proof and extensible in terms of new SQL features and optimization capabilities.

2.3 Runtime

The runtime is at the heart of a query engine. It is responsible for the query evaluation and has to implement data types, functions and relational operators such as joins and aggregations.

In a similar vein to the parsing and planning layers, Firebolt had the choice of building a new query engine from scratch or bootstrapping one from an existing open-source project. Many database companies decided to build their runtime from scratch. Examples include CockroachDB [2], Databricks [5] and Snowflake [10] . We believe that in order to disrupt the data warehousing space as a small startup, it is a better choice to start with an existing codebase and allocate our comparatively limited engineering resources to Firebolt's unique differentiating features.

We had a few basic guardrails when deciding on which project to use as a baseline for our runtime.

To support user-facing, data-intensive applications, we needed a high-performance query engine allowing for low-latency query processing. There are two modern approaches to building high-performance runtimes – vectorization [6] and code generation [18]. We decided that we wanted to build Firebolt on top of a vectorized runtime. While building a low-latency code generating engine is possible, it requires substantial investment into an advanced compilation stack [15, 16]. This increases engine complexity, makes it harder to onboard new engineers, and retain high development velocity. At the same time, code generating and vectorized engines often perform similarly for OLAP workloads [14].

We also wanted the engine to be robust and have basic support for distributed data processing. While there are a variety of interesting academic and experimental open-source query engines, many of those are not production ready or do not support distributed query execution. Starting out with a battle-tested and horizontally scalable engine allowed us to quickly assemble a robust query engine that could process massive data sets.

Alongside our runtime, we also wanted to bootstrap our storage engine, in particular the file format. In order to build a high-performance query engine, it was essential that the storage engine uses a columnar data layout to efficiently support OLAP workloads [1]. The challenges when bootstrapping a query runtime and storage engine are similar to those encountered when bootstrapping a parser and planner. It is possible to pick different systems as a baseline for the storage engine and the query runtime, respectively. However, both components share complex interfaces in order to transfer data between the layers and facilitate data pruning. As such, choosing a single project to supply both components makes it significantly easier to build a high-performance system.

While there are multiple projects to choose as a baseline for the SQL parser and planner, there are fewer options when choosing a

[3]https://calcite.apache.org/docs/powered_by.html
[4]https://github.com/duckdb/duckdb/tree/master/third_party/libpg_query#readme

high-performance, distributed runtime. We quickly narrowed down our choice to ClickHouse [9].

ClickHouse is designed to be fast[5], and these claims are backed-up by benchmarks[6]. ClickHouse uses vectorized query execution, with limited support for runtime code generation through the LLVM. ClickHouse is battle tested and widely used in production environments [7]. Finally, ClickHouse also has its own columnar file format called MergeTree[8]. MergeTree is tightly integrated with the query runtime and allows for efficient data pruning. All of the above reasons made it easy to choose ClickHouse as a foundation for the Firebolt runtime.

2.4 Connecting Planner and Runtime

Section 2.2 presented our decision to use Hyrise as a basis for our query planner. Afterwards, Section 2.3 showed why we chose to base the Firebolt runtime on ClickHouse. This Section outlines how our approach to communication between planner and runtime evolved over time.

Every node in the Firebolt cluster can serve both as query coordinator running parser and planner, and as a runtime worker executing parts of the larger query plan. This is shown in Figure 1. When a query enters the system, it is routed to one of the nodes. This node then acts as the coordinator. After query parsing and planning, it needs to initiate query execution. To achieve this, it needs to transform the planner's optimized LQP into a representation that the ClickHouse-based runtime can understand. The ClickHouse SQL dialect can serve as such a representation. The ClickHouse parser transforms ClickHouse SQL into an internal parse tree that can be executed directly. We used this to quickly build an initial version of cross-component communication. Through a process we called "backtranslate", an LQP within the planner was transformed back to a representation in ClickHouse SQL. This representation then yielded a ClickHouse query plan that conformed to the optimized LQP chosen by the Firebolt planner.

This approach allowed us to get to a working product quickly, but had multiple problems. When connecting planner and runtime this way, a lot of time was spent generating ClickHouse SQL, just to be instantly consumed by the ClickHouse parser again. Even more importantly, a lot of valuable context about the structure of the LQP was lost when going back to a SQL-based representation.

Because of this, we decided to completely replace the backtranslate flow. Nowadays, the Firebolt LQP gets transformed to a distributed query plan directly. This distributed query plan is then disassembled into multiple stages. The coordinator sends stages to different workers within the Firebolt cluster to facilitate distributed query execution. For cross-network communication, we use a custom, protobuf-based serialization format. On the worker nodes, this serialization format is used to assemble the runtime's vectorized relational operators. In the future, Substrait[9] might be a viable alternative to using a custom serialization format. At the time of writing however, Substrait is still undergoing rapid development and is subject to breaking changes.

2.5 Distributed Execution

After choosing ClickHouse as the basis for the Firebolt runtime, we also initially utilized ClickHouse's approach to distributed query processing.

Distributed query processing in ClickHouse works very well for certain shapes of queries. Examples are queries with selective table scans, distributed aggregations on low-cardinality group-by fields, and broadcast joins. At the same time, ClickHouse does not support many important SQL patterns commonly found in data warehousing such as joins between two large relations, aggregations with high-cardinality group-by fields, window functions without granular PARTITION BY clauses, and large distributed sorts.

We decided to move away completely from the ClickHouse distributed execution stack to better serve the data-intensive workloads of our customers. For this, we implemented a new Firebolt distributed processing stack. The optimized LQP returned by our planner is broken up into stages that are connected through shuffle operators. By scaling out, this allows us to execute queries that our original runtime struggled with. Examples include queries with high-cardinality aggregations or joins of two large tables. We will share more about our distributed processing stack in future publications.

Implementing this new distributed processing stack was only possible because we decided early on that our runtime should be a hard fork of ClickHouse. This gives us the flexibility to perform significant refactoring of interfaces, change the overall architecture of the system, and build a runtime that is optimized for the needs of our customers.

3 TESTING

Building software is not just about writing code. It is also about making sure that the code works properly. This is especially important for database systems. Users entrust us with their data and rely on us to produce correct query results. This is not something to be taken lightly.

While we invested heavily into writing our own test cases, we only hand-crafted a small fraction of the test cases that are available elsewhere. Fortunately, when integrating open-source test cases you do not have to choose between different test suites and frameworks. Rather, you can use multiple different ones to combine their own unique strengths.

At Firebolt, we aimed to go for maximum coverage and took test cases from wherever we could. This meant that many test cases did not pass for a variety of reasons. Differences in SQL dialect, differences in features, unspecified behaviour, and much more make it hard to port over test cases from other frameworks and have them "just work". However, integrating with a diverse set of open-source test suites helped us identify real problems. The passing tests make sure we do not regress in functionality or correctness.

3.1 Firebolt Query Verification Framework

Most SQL query test frameworks follow similar patterns – they are based on one or multiple test files outlining a data flow. First off,

[5]https://clickhouse.com/docs/en/faq/general/why-clickhouse-is-so-fast/
[6]https://clickhouse.com/benchmark/dbms/
[7]https://clickhouse.com/docs/en/faq/general/who-is-using-clickhouse
[8]https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree/
[9]https://substrait.io/

SQL DDL queries may be used to define schemata. Then, SQL DML queries hydrate tables with data. Finally, the SQL queries comprising the actual test are run. These are mostly SELECT statements. Their results are checked against pre-defined expected results.

We have built a custom test framework, named "PeaceKeeper", to follow a similar pattern. PeaceKeeper knows how to set up a Firebolt cluster in multiple environments. These can be local runs on a developer machine allowing for fast iteration, CI pipelines, or a powerful distributed SQL cluster in the cloud. PeaceKeeper uses test files with input queries and expected results. We have implemented 2K+ Firebolt specific SQL query test cases.

3.2 Clickhouse Functional Tests

Given that our runtime is based off ClickHouse, it was natural to integrate the 30K+ functional tests provided by ClickHouse[10]. However, since the ClickHouse SQL dialect is very different from Firebolt's SQL dialect, these tests do not pass through our entire query pipeline. Rather, they work directly against the runtime. We are deprecating these test cases, since our distributed execution stack outlined in Section 2.5 is not integrated with the CickHouse SQL parsing layers anymore.

3.3 Postgres Regression Tests

Since Firebolt strives to be as close to Posgres SQL as possible, it made sense to reuse the Postgres test suite. It has 12K+ tests in its regression suite[11]. Not all of the Postgres constructs are supported in Firebolt. However, for the ones that Firebolt does support, we are able to verify that Firebolt behaves in the same way as Postgres. This was done through an automated script that converts the Postgres regression tests into the PeaceKeeper format.

3.4 ZetaSQL Compliance Tests

ZetaSQL has 60K+ tests in its compliance test suite[12]. The distinctive thing about the ZetaSQL compliance tests is that a very large portion of the test cases focus on SQL expressions and individual functions, extensively covering different boundary conditions – something that many other test suites only do in a cursory manner. ZetaSQL's compliance tests are mostly code-based as opposed to file-based. In order to keep our testing tools consistent, we wrote a translator program to capture the test queries in the PeaceKeeper format.

3.5 SQLLogicTest

The pinnacle of functional SQL testing is SQLLite's SQLLogicTest framework[13]. It contains 7M+ test queries (that's 7 million!). We again wrote a script that ports the SQLLogicTests into PeaceKeeper format. A similar approach was taken by DuckDB[14]. Due to the sheer volume of queries, we are still in the process of going over each individual result and sorting out errors.

[10]https://clickhouse.com/docs/en/development/tests/
[11]https://www.postgresql.org/docs/current/regress.html
[12]https://github.com/google/zetasql/tree/master/zetasql/compliance
[13]https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki
[14]https://duckdb.org/dev/testing

4 LESSONS LEARNED

In the following, we want to briefly summarize some of the lessons learned while building Firebolt.

Choosing a Solid Foundation. Especially for query parsers and planners, a wide variety of mature projects can be used as a baseline. While we found Hyrise to be a great foundation and are happy with the choice we made at the time, we would recommend using DuckDB or Calcite for engineers assembling a new system today. Although there are fewer choices for a high-performance distributed runtime, we found ClickHouse to be a robust and extensible foundation. When building a production system, we recommend choosing battle-tested projects as a starting point.

Building in a Single Language. Assembling the system from projects written in a single programming language allows for higher velocity. This is especially true in the early days of a startup when developers might have to frequently switch between different components or build features across the whole stack.

Connecting Different Systems. When choosing different systems for e.g. the planner and runtime, engineering teams need to invest heavily in building clean interfaces between these components. We expect similar effects when basing the parser and planner, or the runtime and storage engine on different projects. Because of this, we recommend choosing as few systems as possible in order to assemble a new database management system.

We only chose two systems written in the same programming language at Firebolt. The effort required to connect them and turn them into a production-ready system was still significant. Moving the systems ever closer requires a concerted effort and is still ongoing. Our new distributed execution stack outlined in Section 2.5 is an example for this. But much work remains to be done. A good example is unifying the type systems across planner and runtime. We believe that not having to integrate widely different projects is the main benefit of building a new system from scratch.

Maybe somewhat surprisingly, the Apache Arrow[15] project did not play a significant role while composing Firebolt's query engine. Many existing open-source projects use Arrow to get data into and out of the system. At Firebolt, all data transfer happens within the distributed primitives of the runtime and when returning data from the runtime back to the user. As a result, we did not need to integrate with Arrow to e.g. facilitate cross-project data transfer. We expect that assembling database systems from different components will become easier as Arrow and related projects find even wider adoption across open-source projects.

Testing the System. We recommend integrating with as many test frameworks as early as possible. While this might seem like a lot of work in the beginning, it pays off in the long run. It allows teams to quickly catch potential compatibility issues in the targeted SQL dialect, as well as subtle differences in the runtime behaviour compared to other systems. However, sorting through test case failures to figure which ones point at underlying issues and which ones are more benign takes a lot of time.

[15]https://arrow.apache.org/

	SQL Dialect and Parser	**Planner**	**Single-Node Runtime**	**Distributed Execution**
Inspired By	Postgres	Calcite		
Based On	Hyrise	Hyrise	ClickHouse	Firebolt
Tested By	Postgres Regression Tests			
	ZetaSQL Compliance Tests			
	SQLite SQLLogicTest			

Table 1: A summary of the open-source components that served as a foundation or inspiration to build and test the Firebolt cloud data warehouse.

5 CONCLUSION

Throughout this paper, we have shown how we used different open-source components as stepping stones to assemble the Firebolt query engine. These components are summarized in Table 1.

The large number of high-quality open-source projects have made it possible for us to build Firebolt in less than 18 months. Going this route as a startup is a smart choice. It allows the organization to quickly converge to a working system. Given the number of mature commercial systems in the space, this allows engineering teams to focus on differentiating features. We are thankful to the open source community, and we intend to contribute back wherever possible.

Assembling a system as outlined in this paper is only the very first step. Building a world-class, high-performance database systems is a marathon and not a sprint – the hard work only starts once a first version of the system is running. For us, deciding to be a hard fork of our open-source roots is essential. It allows for the flexibility to completely redesign the system to better meet our customer's needs. Firebolt is investing heavily into its engineering teams, and large projects are ongoing to improve or rewrite our storage, execution and metadata layers. We look forward to sharing more about these projects in future publications.

ACKNOWLEDGMENTS

We thank the contributors and maintainers of Hyrise and ClickHouse for providing an exceptional foundation for Firebolt to innovate on.

This work reflects significant effort by Firebolt's engineering teams. It would not have been possible without the exceptional work of current and past Firebolt engineers Ariel Yaroshevich, Eldad Farkash, Daniel Heilper, Jonathan Goldfeld, Cosmin Pop, Daniel Ungur, Rares Mironeasa, Ronen Fishman, Joel Shalit, Izik Orgad, Ernest Zaslavsky, Jonathan Doron, Svetlana Messel, Artem Yasynskyi, Vadim Entov, Ivan Koptiev, Boaz Farkash, Elad Dolev, Ori Brotovski, Michael Dessenov, Vlad Davidchenko, Kfir Yehuda, Shahar Shalev, Galit Koka Elad, Oren Wegner, Leonid Bobovich, Itay Israelov, Viacheslav Zubko, Michael Umansky, Arkadi Kagan, Noam Fisher, Yarin Hananiya, Kevin Marr, Cody Schwarz, Gil Cizer, Andrey Kisel, Roman Rustakov, Oleksandr Dudchenko, Francesco Dondi, Ido Kidron, Yuval Boker, Sasha Solganik, David Boublil, Michal Szadkowski, Dany Azriel, James Hill, Imre Palik, Ilya Yakubenko, Nikolay Mizin, Octavian Zarzu, Michael Kravchenko, David Welch, Lorenz Hübschle-Schneider, Dima Sporov, Nir Levy, Maxim Axelrod, Sebastian Kurella, Loic Reyreaud, Leonard von Merzljak, Arnaud Comet, Brit Myers, Jusufadis Bakamović, Tobias Humig, Tal Zelig, Shira Fried, Peter McConnell, Zhen Li, Misha Shneerson, Michael Moursalinov, Aurash Behbahani, Zuyu Zhang, Gracco Guimarães, Andrés Senac Gonezález, Arsenii Krasikov, Bruhathi Sundarmurthy, Branden Pleines, Giuseppe Mazzotta, Anton Perkov, Asy Ronen, Andrei Burago, Alex Hall, Ron Dubinsky, and Ying Li.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280.

[2] Subiotto Marques Alfonso and Rafi Shamim. 2019. *How We Built a Vectorized Execution Engine.* Cockroach Labs. https://www.cockroachlabs.com/blog/how-we-built-a-vectorized-execution-engine/

[3] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 331–343. https://doi.org/10.1145/3035918.3056103

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[5] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy G. Armstrong, David Cashman, Ankur, Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala, Luszczak, Prashanth Menon, Mostafa Mokhtar, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei A. Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 14. https://doi.org/10.1145/3514221.3526054

[6] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. Citeseer, 225–237.

[7] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying Serving and Analytical Data at YouTube. *Proceedings of the VLDB Endowment* 12, 12 (aug 2019), 2022–2034. https://doi.org/10.14778/3352063.3352121

[8] Sid Choudhury. 2020. *Why We Built YugabyteDB by Reusing the PostgreSQL Query Layer.* YugabyteDB. https://blog.yugabyte.com/why-we-built-yugabytedb-by-reusing-the-postgresql-query-layer/

[9] ClickHouse. 2021. *Overview of ClickHouse Architecture.* ClickHouse. https://clickhouse.com/docs/en/development/architecture/

[10] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[11] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 313–324. https://doi.org/10.5441/002/edbt.2019.28

[12] Firebolt. 2022. *The Firebolt Cloud Data Warehouse Whitepaper.* Firebolt Analytics. https://www.firebolt.io/resources/firebolt-cloud-data-warehouse-whitepaper

[13] Lukas Fittl. 2021. *Introducing pg_query 2.0: The easiest way to parse Postgres queries.* pganalyze. https://pganalyze.com/blog/pg-query-2-0-postgres-query-parser

[14] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proceedings of the VLDB Endowment* 11, 13 (sep 2018), 2209–2222.

[15] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *The VLDB Journal* 30, 5 (sep 2021), 883–905. https://doi.org/10.1007/s00778-020-00643-4

[16] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering*

(ICDE). IEEE, 197–208. https://doi.org/10.1109/ICDE.2018.00027

[17] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proceedings of the VLDB Endowment* 13, 12 (aug 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[18] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (jun 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[19] PostgreSQL. 2021. *Frontend/Backend Protocol*. Postgres. https://www.postgresql.org/docs/14/protocol.html

[20] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[21] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). McGraw-Hill, Inc., USA.

[22] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proceedings of the VLDB Endowment* 11, 12 (aug 2018), 1835–1848. https://doi.org/10.14778/3229863.3229871

[23] Jeff Shute. 2022. GoogleSQL: A SQL Language as a Component. (2022). https://cdmsworkshop.github.io/2022/invited.html#invited5 First International Workshop on Composable Data Management Systems.

[24] Octavian Zarzu. 2023. *Firebolt Indexes in Action*. Firebolt Analytics. https://www.firebolt.io/blog/firebolt-indexes-in-action