Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

ChronoSQL: A SQL interpreter for the Chronolog project

Author: **Pablo Pérez Rodríguez**

Supervisor: **Jaime Cernuda, Anthony Kougkas, Xian-He Sun**

Chicago, 08 2022

Contents

List of Figures

List of Tables

1 Introduction

With the continuous growth in the big data industry, the need to ingest, persist, and process high volumes and velocities of data also grows. Amongst some of the data types arising in the big data industry, there is one kind of data usually referred to as streaming data, which is characterized by being generated continuously, and possibly infinitely, by several (sometimes in the order of thousands) sources. Traditional big data platforms have been found to being incapable of handling such volumes and velocities of data. To face this scenario efficiently, many platforms, usually referred to as streaming engines, have emerged in recent years. Well-known production-ready platforms include systems such as Apache Kafka and Amazon Kinesis, amongst others.

This work focuses on ChronoLog, a state-of-the-art time based streaming platform. ChronoLog is adistributed, shared, and multi-tiered log store that has been designed to handle more than a million tail operations per second [6]. It has been engineered by the Scalable Computing Software Laboratory at the Illinois Institute of Technology.

One of the issues faced by this kind of platforms is that they usually offer a low-level API specific to the individual platform. This APIs provide, in most cases, low-level operational semantics that can be powerful in the hands of an experienced user. Despite that, they induce a steeper learning curve for the new users of these systems, and require some initial adaptation when moving from one system to another, as APIs diverge among systems.

Yet, the Big Data industry has a standard query language, SQL, that has been used to fetch and manage data for many years. An extensive amount of users have expertise on this language, making it an easy and shareable entry point for all of these platforms. As such, the ability to query these systems with SQL has been explored extensively for pure data streaming platforms (amongst other systems) as this opens the door for new users to use the platforms.

In this work, we aim to explore the feasibility and requirements of implementing a SQL querying layer on top of time-enabled log storage systems, such as ChronoLog. We shall base our design on a thorough exploration of the current state-of-the-art solutions in the industry. And present ChronoSQL, a SQL interpreter designed for the ChronoLog platform, and any other time-enabled log storage that may arrive in the future.

ChronoSQL is responsible for five core tasks:

- Reading, validating, parsing, and transforming a SQL statement into a query tree
- Interpreting the query tree
- Translating from abstract operations to ChronoLog API operations to fetch the corresponding data
- If needed, transforming the fetched data
- Returning the results of the query

ChronoSQL is a user-space library that focuses on time operations and capabilities. As ChronoLog was designed around time, and makes efficient use of timeseries data, ChronoSQL should take advantage of this to offer new functionalities that make the most use of timestamped data.

The rest of the report is structured as follows: Section 2 presents a background on similar systems and existing solutions; Section 3 shows the design and development of the library; Section 4 shows the evaluations performed on this project; finally, Section 5 shows the conclusions of the project and some future lines of work that could be added to the ChronoSQL library.

2 State of the Art

The continuous growth in real-time information generation has created several challenges to the computer science community. These very high volume data need to be ingested, stored, and usually re-routed to different types of receivers that will perform some kind of computation or storage with the data. To support this growth, there have been several development efforts in the field to create platforms that can deal with these situations. Data streaming systems introduce a novel approach to this problem by processing data in real time as it flows through the system. This allows these platforms to collect and deliver high volume data from sources to the destination applications in an efficient way[7]. Among these platforms, a well known example of such a system is Apache Kafka, originally developed at LinkedIn. It has been designed to handle the billions of events that happen daily in their social network as a scalable publish-subscribe messaging system. It is one of the most used streaming platforms in industry [8]. Other examples of such systems are Amazon Kinesis, Microsoft EventHubs, and RabbitMQ. As these systems have existed for a while, event processing platforms have also emerged to offer an additional way to extract insights from the data stored in event streaming platforms. For example, Apache Spark has an streaming extension, Spark Streaming, that enables stream processing from multiple sources (Kafka, Kinesis, among others). Some systems have also been developed specifically for this purpose, such as Apache Flink, a computation system that processes streams of data in real time.

What all these systems have in common is that they offer their own APIs to perform streaming operations. This is, each system provides a series of operational and syntactic distinctions that, even though they have some similarities, require users to get to know the specific syntax and particularities each system offers. Additionally, data streaming is a programming paradigm that works with DAGs (Directed Acyclic Graphs), link jobs, etc., so they are also a shift on paradigm even for those who know how to program. This makes these systems harder to acquire by certain types of potential users that do not have such a background, even though they provide powerful computation capabilities.

To alleviate this issue, some platforms have decided to offer SQL interfaces to allow users to query streaming data. SQL is a standardized programming language used to retrieve and manage data, originally from relational databases. It is one of the most popular programming languages [9], and is also very popular among non-developers, this is, people that are not familiar with programming, because of its ease of use and the capabilities it provides. This new SQL layer provides an easier way to query data from streaming platforms. It makes it simpler for someone to start extracting information than getting used to a new API, so this makes the adoption process simpler.

The purpose of this section is to analyze some of these SQL streaming solutions in depth, comparing their approach to parse and analyze SQL queries, their architecture and internal design decisions, and the functionalities they offer.

2.1 Event streaming platforms (log stores)

As already mentioned, data in industry is being produced at extremely high rates. Multiple sources such as sensors, networks, customer actions, transactions, etc., produce data continuously, in some cases reaching millions of data instances per day.

As this is real-time data, its value can quickly degrade, so it is very important to extract insights and information from these data sources almost immediately. For example, sensor reading data may need to be interpreted as soon as it is generated to detect anomalies and act accordingly. Additionally, some of this data may also be useful in the future, to analyse past trends and predict, for example, user behaviour.

Traditional relational databases have shown to be ineffective for this task. They were not originally designed for this kind of dataflows, and their data model is significantly different from the streaming approach. In traditional databases, only the current state of the data is relevant, while in the latter ones, historic data processing is required. Also, the traditional model does not support the continuous queries that streaming data require to allow real-time analysis of the data that is ingested into the system [10].

As a consequence, in recent years there has been a development effort to design systems that are capable of

- Ingesting huge volumes of real-time data that are produced at very high rates
- Providing high data throughput
- Allowing real-time analysis as soon as the data is received
- Also allowing historical data processing that comprises very large amounts of streaming data

One of these systems is Apache Kafka. It was developed by LinkedIn to handle all the data their social network generates. Linkedin was in a situation where they had extremely high volume event data being generated constantly, and they needed not only to store these data, but also to process and deliver it to diverse subscribers. For example, Kafka is used to power the "Who's viewed my profile?", in order to recollect all the profile visualizations and feed it to the LinkedIn app to show it to users [8].

Kafka has been architected as a publish-subscribe messaging system implemented as a distributed and partitioned commit log [11]. Its design differentiates between *producers* (processes that generate messages) and *consumers* (processes that consume messages). Consumers ingest messages by subscribing to *topics*, which are some sort of categories where producers insert their messages to categorize them. Kafka is deployed on a set of servers, called brokers, that are responsible for storing the messages for each topic in partitions, and delivering them to the corresponding consumers.

Additionally, Kafka (and similar platforms) provide stream processing functionality. Stream processing consists on producing low-latency outputs from a sequence of input data, performing some computations on the continuously arriving data. For this purpose, LinkedIn developed Samza, a distributed stream processing framework that is built on

top of Kafka, and that is used by SamzaSQL, one of the SQL platforms we will review later.

Another popular streaming platform is Amazon Kinesis, also developed to handle the real-time processing of massive scale data at Amazon. Kinesis can collect hundreds of terabytes of data per hour from a large number of different sources, and process them to feed computed data to real-time dashboards, alert systems, recommendation engines, etc [12]. These data can also be redirected to storage systems such as Amazon S3.

Other examples of streaming platforms include Azure Event Hubs, developed by Microsoft with a similar approach to Kinesis, and RabbitMQ, an open source alternative that is also used by big companies, such as T-Mobile and Adidas [13].

2.2 SQL on streaming

In order to retrieve data from these systems, some platforms have decided to offer SQL interfaces to allow users to query streaming data. As it is one of the most popular programming languages among developers [9], but also for those that do not have a programming background, it has become a way for platforms to support a high number of potential users. With this approach, the learning curve is reduced compared to learning a completely new API, and thus the adoption process is easier for new users.

In this section, we will explore four streaming SQL services:

2.2.1 SamzaSQL

SamzaSQL is a SQL streaming platform originally developed by LinkedIn as part of their stream processing framework, Samza (now listed under the Apache foundation as Apache Samza). It is built on top of Apache Kafka, and adds more stream processing capabilities to it [14].

Using this streaming processing framework, they added the capability of processing streaming and batch data using a common standard language, SQL, creating the library SamzaSQL. Basically, this library converts a SQL query to a Samza job that performs the computations on the data. For this purpose, SamzaSQL uses another Apache library, Calcite [1].

Apache Calcite is a query processing system that performs query parsing, validation, planning and optimization tasks on the received SQL queries [15]. It is employed by SamzaSQL to perform these tasks, and then the output produced by Calcite and converts it to Samza model to pass it to the Samza system and process the data.

In figure 1 we can observe the high-level design of SamzaSQL. As we can observe, SamzaSQL provides a SQL shell that utilizes the Samza YARN client to submit jobs that will later be executed by Samza, Zookeeper for storing metadata and internal configurations, and, as already mentioned, Calcite to parse and plan SQL queries [1].

Even though it is not shown in the figure, SamzaSQL relies on Samza for most of

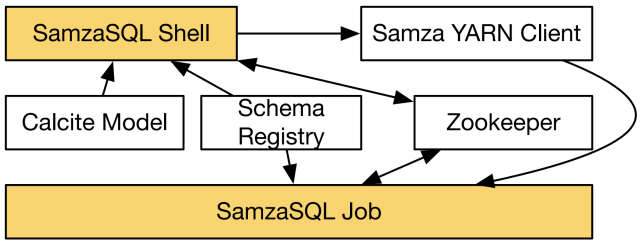

Figure 1: SamzaSQL architecture (extracted from [1])

the data processing that is carried out. After a query has been processed by Calcite, SamzaSQL performs some additional optimizations and transforms the query plan into a Samza job, that is passed to the Samza system to obtain the results.

2.2.2 ksqlDB

ksqlDB takes a little different approach to the SQL streaming providing a complete streaming database package, in the sense that it does not offer just a layer of SQL abstraction like SamzaSQL does, but rather a complete platform that contains all the necessary components in a single deployment [2].

It has been developed by Confluent, a company created by the original developers of Apache Kakfka [16]. It can be scaled out as needed by deploying several engines running in parallel, and it gives the possibility of scaling out a cluster even while operations are being executed [2].

It has been tailored to work with Apache Kafka, and as a consequence, it only works with a Kafka deployment, as opposed to SamzaSQL. Its high-level deployment architecture is available in Figure 2.

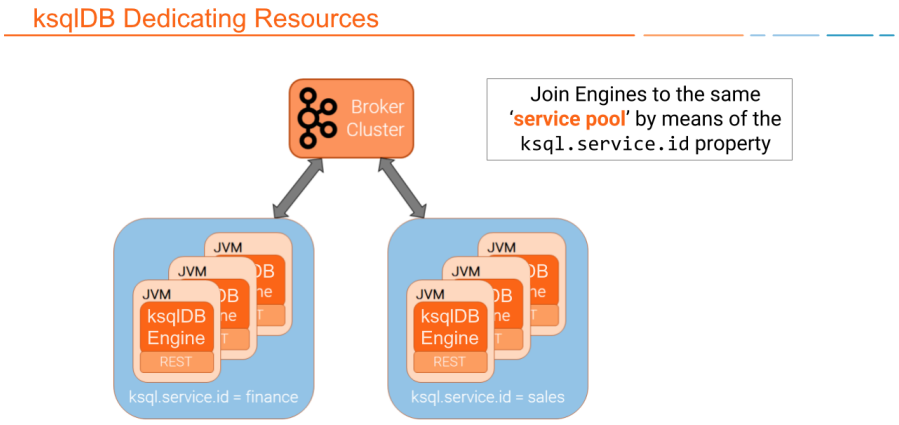

Figure 2: ksqlDB architecture (extracted from [2])

As it is shown, a ksqlDB cluster does not need a master node or any additional coordination mechanism, so every ksqlDB node acts an equal peer. Additionally, ksqlDB not only allows to have several instances running in the same cluster, but it also provides the functionality of *services*. Services offer a way to have more than one ksqlDB cluster

connected to the same Kafka deployment easily, which can be used, for example, to feed data to more than one application, each of which uses its own set of resources and does not affect the others [2].

2.2.3 Materialize

Materialize takes a similar approach to ksqlDB, offering a full database-like solution for streaming data —actually, also from data stores and databases. It is based around the concept of materialized views, queries that are cached and incrementally updated so that whenever they are run, the results can be obtained very quickly [17].

Their goal with this is to quickly provide answers to questions that are frequently asked, for example to feed data into dashboards that are updated routinely [3]. For this, they have built the core of their system around the Differential Dataflow, a data-parallel compute engine that allows to update materialized views very efficiently as soon as the data arrive in the system.

It is worth noting that, even though Materialize have built their system with a very high emphasis in materialized views, it is not something unique to this platform. ksqlDB also provides support for this kind of queries, even though the Differential Dataflow framework developed specifically to handle them may provide more efficient updates to the views. More on this will be discussed in section 2.3.

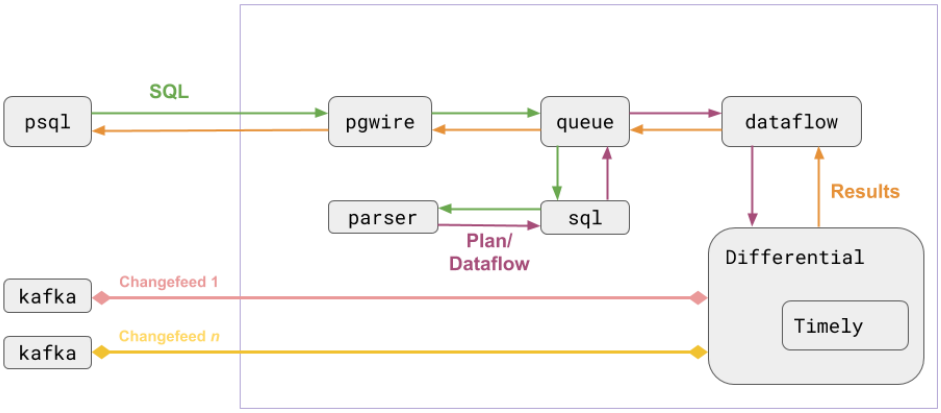

Figure 3: Materialize architecture (extracted from [3])

Materialize's architecture can be observed in Figure 3, where the main components that shape the platform are presented. It uses *pgwire*, Postgres wire protocol to communicate a SQL client with the system. The SQL statements received are then passed to a queue that holds incoming queries until they are passed to a parser. Once that is done, the plan is sent over to the Differential Dataflow system which handles the computation of the queries, retrieves data from data sources, and sends the results back so that they are received by the client (after passing again through the chain of components) [3].

2.2.4 Facebook Presto

Finally, the last system that is going to be covered is Facebook Presto. Presto is a distributed SQL query engine developed by Facebook (even though it is now an open source project) that aims to provide a way to use the SQL language to extract data from almost any data source. In contrast with the platforms that have already been presented, Presto has not been designed with the only purpose of working with streaming data, but has rather taken a broader approach and it is capable of also querying data from traditional databases, Hadoop environments, and NoSQL systems (even from several of these sources with just one query) [18].

Presto was designed with the main goal of offering the highest performance. It is capable of running hundreds of memory, I/O, and CPU-intensive queries concurrently, scaling to thousands of nodes if needed [18]. It is used by Facebook to handle many use cases that require very large deployments to process huge data volumes.

It provides easy integration with streaming sources such as Kafka and Kinesis. Nevertheless, as it was designed not only for this kind of sources, it does not provide all the streaming functionality the previous systems do, such as materialized views.

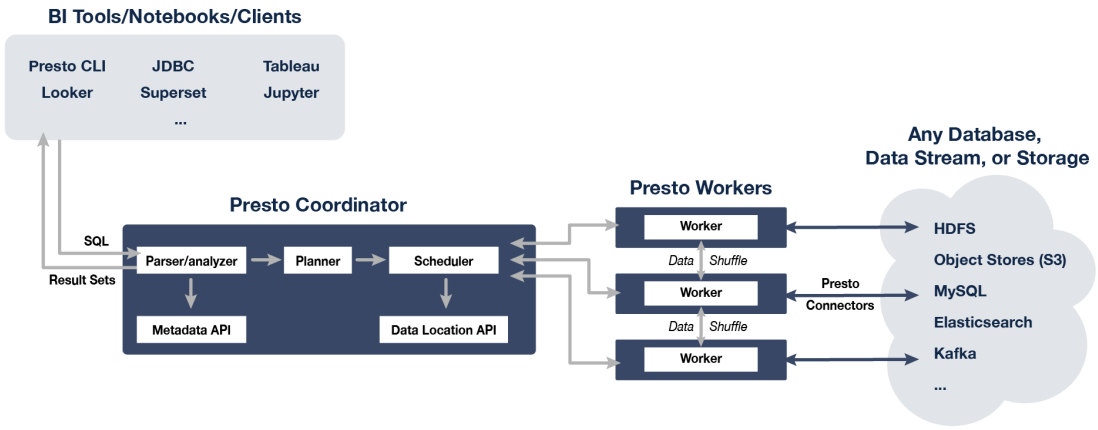

Figure 4: Presto architecture (extracted from [4])

Figure 4 provides an overview of the internal architecture of Presto. Basically, Presto employs a master-slave architecture, with one coordinator in charge of handling the query parsing, optimizing and plan generation, and several workers that execute individual tasks assigned by the coordinator. These workers are connected to the data sources and perform the corresponding computations on the pieces of data they extract. Each of these individual results are sent to the coordinator, that merges all of them and returns the final result to the client.

2.3 Comparison of the explored solutions

In this section, a comparison of the solutions introduced in Section 2.2 will be presented. For this purpose, different functionalities will be analyzed, and a final comparison of all the features will be provided.

2.3.1 Streaming queries

One key feature for streaming SQL engines is what is usually referred to as *streaming queries* (each system may provide its own naming, for example ksqlDB calls them *pull queries*, but we will refer to them as *streaming queries* for the remaining of this report).

A streaming query is a type of query that is running in the background and emits changes as updates are performed on the target tables/streams. It can be seen as a subscription to real-time changes. Internally, it has a mechanism to detect any change made to the data sources involved in the query, these changes are then (efficiently) processed, and the result is sent back to the client [19].

These queries have several applications, and can be very useful for example to power real-time dashboards in a very efficient way because the client does not need to be continuously polling the server for updates, and also because these updates do not need to be recomputed.

From the platforms analyzed, only Presto does not support streaming queries. This is probably because the other platforms are tailored for the streaming environment, while Presto is a much more generic system that was adapted, through *plugins*, to work with streaming platforms.

2.3.2 Materialized views

Another interesting feature that is not only present in streaming SQL, but makes even more sense for these systems due to the high number of events recorded in streaming platforms, is the concept of materialized views.

These views are a special kind of queries that cache the results of the query and update these results continuously to provide quick access to them [17]. Usually, in traditional databases, these computations have to be done at set intervals (for example, every 30 seconds), so there will be situations where no data needs to be processed, and others where a large piece of data needs to be updated. Additionally, some databases do not have an optimized way of computing updates, so sometimes a complete computation of the entire data needs to be done.

Nevertheless, streaming SQL engines provide certain optimizations that make these views very efficient. From the platforms studied, only ksqlDB and Materialize provide these views, so we will check in detail how each of them handles materialization.

In the ksqlDB platform, whenever a new event is integrated, the state of the view evolves by applying the query to the new event. This is, a view is never completely recomputed when new events arrive, but it is incrementally updated so the processing is done efficiently [2].

This platform differentiates between materialized tables and non-materialized tables. The former ones can be queried like any table, and queries against them are very efficient because data will be already computed. On the other hand, non-materialized tables are not allowed to be queried, because according to ksqlDB, this would be very inefficient.

The other system that supports materialized views, Materialize, has built a system just to handle these views, the Differential Dataflow. It is a data-parallel compute engine that was designed to efficiently perform computations on large amounts of data and maintain the computations as the data is updated [20].

When a materialized view is created, a *dataflow* is generated. A dataflow is a series of instructions that specify how to act when data arrives to the system [3].

It is also worth mentioning that the state of these views is stored in memory for both systems. ksqlDB uses an external library, RocksDB, for this purpose, while Materialize handles the storage on its own.

2.3.3 Query management

Even though these systems are quite similar, the way they process a query after receiving it may differ among them. In this section, the query management and execution of each platform is going to be analyzed:

2.3.3.1 SamzaSQL

Figure 5 shows the overview of the query planning process carried out by SamzaSQL [1].

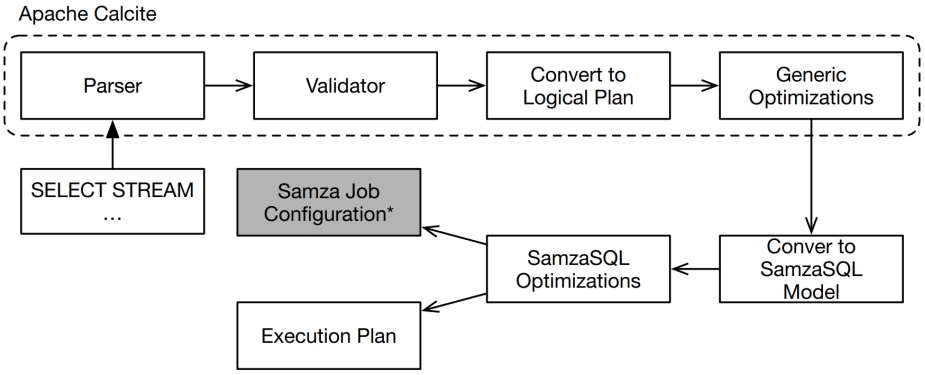

Figure 5: SamzaSQL query planning (extracted from [1])

As it can be observed, Calcite is in charge of parsing, validating the input query, generating the logical plan, and applying some optimizations to this plan (for this, it uses a set of predefined generic rules, plus some specific ones provided by SamzaSQL).

After that, the optimized logical plan is converted to the SamzaSQL model, this is, the physical plan that can be interpreted by Samza. In a way, this step is like a translation from the generic SQL logical plan to the set of Samza operators. This physical plan is also optimized by SamzaSQL before proceeding to the following phase.

There is also a step where some Samza configuration is generated from the physical plan, producing some metadata that is stored temporarily in Zookeeper to make it available so that it can later be accessed.

Finally, all this information is packaged as a execution plan that is submited as a Samza job to the Samza system using Samza's YARN client.

2.3.3.2 ksqlDB

Figure 6 shows how ksqlDB transforms a SQL query into a sequence of Kafka Streams operators.

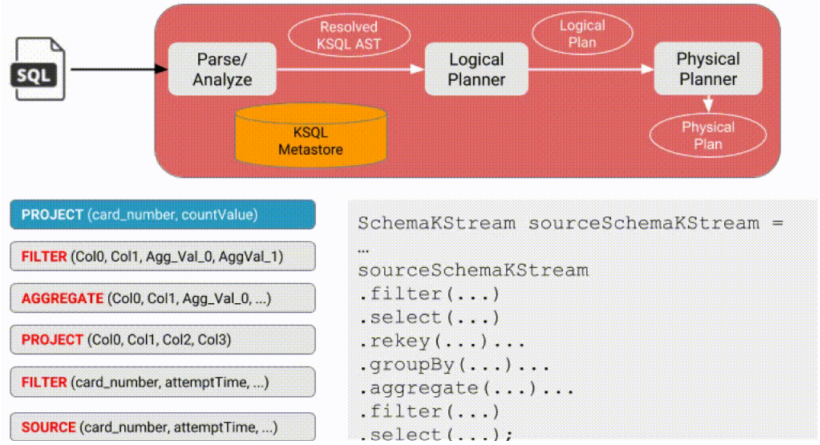

Figure 6: ksqlDB query planning (extracted from [2])

The steps taken are the following [2]:

1. First of all, a stream or table from an existing Kafka topic has to be declared before fetching any data.
2. Then, a SQL statement is issued to the system.
3. ksqlDB parses the received query statement into an Abstract Syntax Tree (AST).
4. The AST is analyzed.
5. The analyzed AST is used to create a logical plan by the logical planner (an example of how a logical plan would look like is shown in the bottom left of Figure 6).
6. The physical planner uses the logical plan, optimizes it and generates the physical plan (shown at the bottom right of Figure 6).
7. Finally, ksqlDB uses the physical plan to run a Kafka Streams application.

2.3.3.3 Materialize

Figure 7 shows the top-level view of how a query is handled. Unfortunately, Materialize does not provide a very detailed information about how it processes a query like the other platforms analyzed do (or it has been impossible to find), so the process explained in this section is a little less detailed than for the other systems.

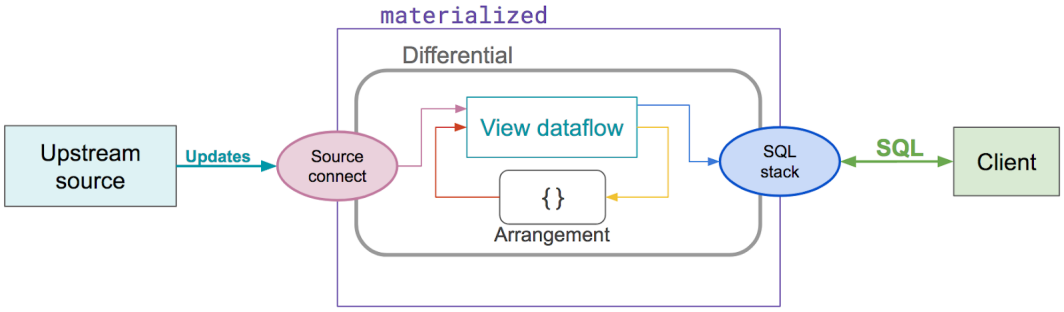

Figure 7: Materialize query management (extracted from [3])

After a user has created a materialized view, Materialize parses the statement and creates a Differential dataflow for it. Once a view has been materialized, any new or updated data from the sources that have beend declared in the view triggers the differential process, which responds to changes in data to update the materialized results.

Whenever a *SELECT* query arrives in the system to query data from the materialized view, the system can directly respond with the already computed data, instead of fetching it from the original data source.

2.3.3.4 Presto

Presto's query execution overview is available in Figure 8. The steps carried out are the following [18].

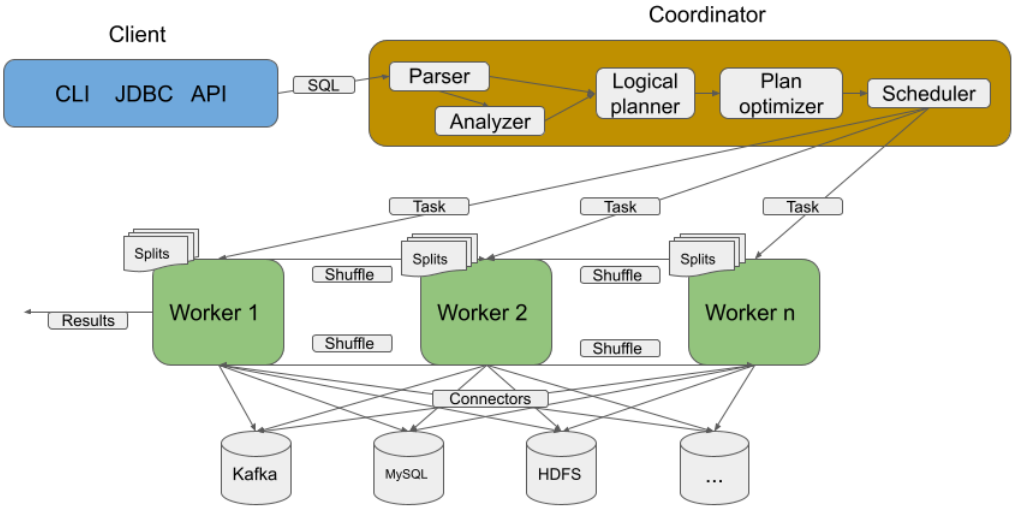

Figure 8: Presto query execution (adapted from [4])

1. First, the client submits a SQL query through one of the available clients (a CLI, the JDBC connector, or a REST API) that is received by the coordinator.
2. Then, it uses an ANTLR-based parser to transform SQL into a syntax tree used by the analyzer.

3. A logical planner uses the output from above to create a tree-shaped plan, where nodes are the operations needed to be carried out, from a highly abstracted viewpoint.

4. The plan optimizer maps the logical plan into a more enhanced version that contains more information about each task and how to efficiently implement it.

   It analyses parts that can be executed in parallel among nodes. These parts are called *stages*, and each stage contains a series of *tasks* that are the pieces of work distributed to the workers.

5. The optimizer also identifies sections inside a stage that can be executed in different threads of a single node. This leads to significant speedups, and is more efficient than inter-node parallelism (less latency overhead, better shared memory).

6. Tasks are then executed by workers, retrieving information from data sources through *connectors*.

7. Finally, the individual results are sent back to the coordinator, who is responsible for merging and returning them to the client who issued the query.

2.3.4 Syntax extensions

In this section, the syntax provided by each of the systems will be analyzed. Even though they are all SQL engines, they provide certain extensions that are interesting for the streaming environment, and we will focus on them for each platform. Note that we will not go through all the SQL operators, but rather those that are interesting from each platform. As SQL engines, a basic set of the SQL syntax is assumed for them.

2.3.4.1 SamzaSQL

The following are the most interesting streaming operators SamzaSQL provides [21]:

- ***STREAM* keyword**: this keyword, used after a *SELECT* statement, tells the system that the client has requested a streaming query, so it should keep processing data when received and output it.
- **Window functions**: SamzaSQL provides support for hopping, tumbling and sliding windows.
- **Joins**: all the stream-to-stream, stream-to-relation, and relation-to-relation joins are supported.

2.3.4.2 ksqlDB

The following operators are offered by ksqlDB and are relevant in the streaming environment [2]:

- ***EMIT CHANGES* keyword**: it is used to specify a streaming query, and must be included at the end of the SQL statement.
- **Window functions**: hopping, tumbling and sliding windows are supported by ksqlDB.
- **Joins**: all the stream-to-stream, stream-to-relation, and relation-to-relation joins are supported. Nevertheless, stream-to-stream joins are only allowed when combined with a window function.
- ***LATEST_BY_OFFSET*** *operator*: this operator returns the latest value (the one with the largest offset) for a specified column. It is used in aggregations.
- ***EARLIEST_BY_OFFSET*** *operator*: this operator returns the earliest value (the one with the lowest offset) for a specified column. It is used in aggregations.
- ***TOPK*** *operator*: this operator returns the top K values for a given column and window. It is used in aggregations.
- ***HISTOGRAM*** *operator*: returns a map containing the distinct values for a given column mapped to the number of times each one occurs for the given window. It is used in aggregations.

2.3.4.3 Materialize

These are the more relevant operators provided by Materialize [3]:

- ***TAIL* operator**: this operator is a variation of the common SELECT operator that works like a streaming query, updating the result as new events are received.
- **Window functions**: Materialize provides support for hopping, tumbling and sliding windows.
- **Joins**: all the stream-to-stream, stream-to-relation, and relation-to-relation joins are supported.
- ***LATERAL* subquery**: Materialize provides a different type of subquery that allows subqueries on the right-hand side of a JOIN to access columns defined in the left-hand side. Additionally, this operator allows to group rows by certain key and obtain the top k elements within each group.

2.3.4.4 Presto

Regarding Presto, again, it does not provide any streaming functionality, so its syntax is limited to normal SQL language.

Features	SamzaSQL	ksqlDB	Materialize	Presto
Streaming queries	Yes	Yes	Yes	No
Relational queries	Yes	Yes	Yes	Yes
Data manipulation statements	Yes	Yes	Yes (data is not persisted)	Yes
Materialized views	No	Yes	Yes	Trino
Queries on streams	Yes	Yes	No	Yes
Cluster mode	No	Yes	No (yet)	Yes
External platforms	Zookeeper Calcite YARN	RocksDB Kafka topics Zookeeper	psql CLI pgwire	No
Hopping and tumbling windows	Yes*	Yes	No	Yes
Sliding windows	Yes*	Yes	No	Yes
Stream-to-stream joins	Yes*	Yes (windowed)	Yes	Yes
Stream-to-relation joins	Yes*	Yes	Yes	-
Relation-to-relation joins	No	Yes	Yes	-
User-defined functions	Yes	Yes	No	Yes
Pluggable to different sources	Yes	No	Yes	Yes
Service/library	Library	Service	Service	Service

Table 1: Feature comparison table. Features marked with an asterisk mean they have not been implemented yet, but are under development and planned to be added.

2.3.5 Feature comparison

In this section, we provide a general comparison of many features that are relevant for the SQL streaming environment, pointing out which platforms provide which features. Table 1 includes all the features that have been selected, and whether each platform provides them or not.

As it can be observed, this table summarizes most of what has been discussed in Section 2.3 (streaming queries, materialized views, window functions, etc.), and adds some other comparison points worth mentioning:

2.3.5.1 Data manipulation statements

One thing about SQL is that it does not only provide functionality to retrieve data, but also to manage it. For this purpose, there are certain statements, such as *INSERT* and *DELETE*, that allow users to modify the records of a data source.

Both ksqlDB and SamzaSQL support *INSERT* statements into a stream, but not update and delete operations. This is mostly because they were designed to work with log stores, and logs are immutable, so only the append operation would make sense [1].

Materialize, on the other hand, provides an *INSERT* operator, but the data is never actually persisted (it is only made to the in-memory representation of a data source). Therefore, whenever a Materialize instance is restarted, all the data that had been inserted is lost [3].

This platform also supports *DELETE* and *UPDATE* operations, but similarly to the insertion, these updates only remain for as long as the modified table is kept in memory.

Finally, Presto is the platform that provides more data manipulation functionality. *INSERT* and *DELETE* operators are fully supported, like in any other database, but there is no *UPDATE* operator.

2.3.5.2 Queries on streams

An interesting issue is whether these systems allow to query a stream (data source) directly or not. ksqlDB, SamzaSQL and Presto support this, while Materialize only allows to *SELECT* from materialized views or materialized sources [3] and not directly from streams of data.

2.3.5.3 Cluster mode

As these systems usually work with very large volumes of data, the processing burden associated with dealing with these amounts of data can be very high. Because of this, some of these platforms offer different deployment options to allow users to scale their installation and perform more advanced computations.

Presto is quite well-known for being designed for very high performance, and for this purpose it can be deployed on a large cluster (reaching thousands of worker nodes) to handle complex queries on very large amounts of data, scaling the number of worker nodes as needed [18].

KsqlDB also allows to deploy several instances to scale a cluster as needed. As mentioned before, ksqlDB clusters do not have a master node, and server instances can be added even in the middle of an operation.

On the other hand, SamzaSQL and Materialize do not give support for a cluster deployment, even though Materialize's documentation mentions that this functionality is under active deployment, so it is likely that a future version will bring it.

2.3.5.4 Use of external platforms

Some of these platforms make use of additional systems to perform their operations.

SamzaSQL uses these systems [1]:

- ***Apache Zookeeper*** is also used by SamzaSQL. In this case, it is mostly used to share metadata and configuration information between the query planner and the streaming tasks SamzaSQL generates.
- ***Apache Calcite***, as mentioned above, is a query processing system that performs query parsing, validation, planning and optimization tasks on the received SQL queries [15].

- ***YARN*** (Yet Another Resource Negotiator), a very well known resource manager for the Hadoop ecosystem, is used in this case by SamzaSQL to submit streaming jobs to Samza, which under the hood uses YARN as resource manager.

ksqlDB uses the following external platforms [2]:

- ***RocksDB***, a key-value store, to maintain the local state of a node. This state consists primarily on the current state of a table (materialized view), so that it is quickly accessed when needed.
- ***Apache Zookeeper*** is a service designed for highly reliable distributed coordination of cloud applications [22]. In ksqlDB it is used to handle cluster metadata and help coordinate all the running instances.
- ***Apache Kafka***. Finally, ksqlDB also makes use of Kafka, not only as a data source, but also to store internal information. Whenever a stream or table is declared, ksqlDB creates a Kafka topic. In addition, aggregations and similar operations, if materialized, require to keep a changelog in a Kafka topic, so if the in-memory cached version is lost, it can be quickly recovered.

Materialize utilizes *pgwire*, the PostgreSQL wire protocol, to connect the Materialize system with its SQL interactive interfaces (either *psql*, PostgreSQL's terminal front-end, or *mzcli*, Materialize's own client, which is basically a wrapper of the *pgcli*, a PostgreSQL command line interface).

Finally, Presto works a standalone platform, with no external dependencies on other systems or platforms.

2.3.5.5 User-defined functions

User-defined functions allow to extend the functionality of a platform with specific needs a user may require. Of the platforms analyzed, only Materialize does not provide support for adding custom functions to the system.

2.3.5.6 Pluggable to different sources

One characteristic of some of the platforms evaluated is that they can be connected to different data sources. While ksqlDB has been designed to read data only from Apache Kafka, the other platforms allow to query multiple data stores.

Materialize can read data from Apache Kafka, Amazon Kinesis and local files, even though the Amazon Kinesis connectivity is still in preview mode. SamzaSQL gives more flexibility, allowing users to set up connections with Apache Kafka, AWS Kinesis, Azure EventHubs, ElasticSearch, and Apache Hadoop thanks to the Samza lower level. In addition, it is in theory possible to read data from custom sources, according to their documentation [21].

Finally, Presto is the platform that provides the highest connectivity support. It allows to read data from a very large range of data sources of many kinds (relational databases, NoSQL databases, streaming data stores, Hadoop data warehouses, etc.). It is also extendable through *connectors*, that specify an API that allows any data source that implements it to be used by Presto [4]. And, one of the biggest advantages of Presto is that it can fetch data from more than one data source at the same time, so a Presto cluster can be connected to PostgreSQL and Kafka at the same time, and read data from both seamlessly for the user executing the query [18].

2.3.5.7 Deployed as a service or library

An important difference among these platforms is the way they can be deployed. There are primarily two ways to develop these kind of applications, as services or as libraries. The first one will be deployed on a separate server and can be accessed through a protocol such as HTTP, while the latter one is directly integrated into the deployment of the system that makes use of it [23].

Among these systems, only one, SamzaSQL, has been developed as a library. It is contained inside the Samza platform, and can be executed once a Samza instance is deployed without any further deployments [24].

The rest of the platforms evaluated were designed as services and as such, they are deployed as standalone instances.

2.3.6 Specific optimizations

Each of these platforms has taken some design decisions that differentiate their approach from each other. This section summarizes some of these decisions and explains why they are important and how they can affect the platform execution.

2.3.6.1 SamzaSQL

As mentioned in previous sections, SamzaSQL uses the Apache Calcite library to parse, validate and optimize the SQL statements received. Therefore, even though SamzaSQL defines some custom planner rules that extend those designed by Calcite, it heavily relies on the planning and optimizations carried out by Calcite which are probably less specific than if they were developed ad hoc for the system.

This platform also used another library, SqlLine, to build a custom shell, the SamzaSQL Shell.

2.3.6.2 ksqlDB

Some of the optimizations ksqlDB has implemented are the following [2]:

- **Cluster deployment without master node**. Every node has the same responsibilities, and a cluster can be scaled out even while an operation is being executed.
- **RocksDB data storage**. As already mentioned, RocksDB is used by ksqlDB to maintain the state of aggregations in memory, so that common queries can be run quickly without needing to fetch data from disk.
- **Kafka topics to store sources information**. Each stream and table defined in ksqlDB will have an associated Kafka topic that is used to store metadata and useful information of the stream/table.
- **Kafka topics to persist materialized state**. Apart from storing materializations in memory, ksqlDB persists a compacted changelog topic that allows to recover the state of a materialized view applying the changelog from the beginning.

2.3.6.3 Materialize

The main optimizations Materialize has made are the following [3]:

- **Differential dataflow**: As already explained, the Differential Dataflow is a data-parallel compute engine that allows to update materialized views very efficiently as soon as the data arrive in the system. It is one of the most important elements of Materialize, and allows to query data very quickly, and to maintain the state of the materialized data updated in a very efficient manner.
- **Join optimizations**: in order to reduce the memory footprint (and avoid to repeat computations), Materialize makes heavy use of indexes, maintaining zero intermediate results in materialized views. For this, almost every primary and foreign key needs to be indexed, which means the initial storage and computation overhead is quite high. Nevertheless, Materialize states in their documentation that, for most cases, this initial overhead is compensated by the reduced per-query cost that allows to work with a quite large number of queries concurrently without exhausting the system.
- **Compaction**: Materialize only stores materialized views in memory. To prevent exceeding the memory requirements, it has a process called *compaction* that is in charge of periodically compacting data, folding historical updates. Here there is a trade-off between historical detail and resource usage. Compacting too often will use less memory, but some historical detail will be lost; a larger compaction window will keep more information, but the memory usage will be higher. Thus, choosing the right compaction intervals will be an important decision when using this system.

2.3.6.4 Presto

Presto's optimizations differ a little bit from the streaming platforms we have seen, due its nature. These are some of them [18]:

- **Master-slave architecture**: work is split between the coordinator of the cluster and the worker nodes as follows. The coordinator is in charge of handling the query parsing, optimizing and plan generation, and the tasks that are generated in these steps are then assigned to several workers that execute them.
- **Extensibility**: Presto can be connected to custom data sources as long as they expose an API specified by Presto.
- **Query data from multiple sources simultaneously**: as already stated, one of Presto's main advantages is the possibility of extracting data from more than one data source at the same time, with just one query so the user does not have to be aware of how data is distributed internally.
- **Resource management**: one single Presto cluster can execute hundreds of queries concurrently thanks to its fully-integrated resource management system, that maximizes the use of CPU, IO, and memory resources.
- **Parallelism**: Presto has been built for very high performance, and for this purpose it was designed to make use of parallelism as much as possible. It optimizes queries so that as many parts as possible can be executed on multiple nodes in parallel, and the optimization process also seeks to identify parts of the individual tasks assigned to a specific node that can be performed in several threads.
- **Early results**: For certain query shapes, Presto is capable of returning results before all the data is processed. This is because worker nodes share data as soon as it is available, so early results can become available almost as soon as they have been processed.
- **Lack of fault tolerance**: Presto does not have any built-in fault tolerance system. Therefore, worker node failures cause all the work performed on that node to be lost, and coordinator failures make the whole cluster unavailable. Nevertheless, it was a decision made to minimize additional overhead that would reduce the overall cluster performance, and according to [18], the expected value of a fault tolerance system is unclear given its cost.

2.4 Time Series Databases

One of the applications of streaming databases (and thus, of streaming SQL engines), is to store and fetch measurements or observations of events as a function of the time at which they occurred, this is, data recorded as a time series [25].

There is a growing number of scenarios where storing this kind of data is useful. Flight data, electrical activity in the brain, rainfall measurements, heartbeats per minute, and stock price changes, are just a couple of examples of time series data.

Storing data like this can be very useful to recognize patterns and trends, because it is possible to observe how a certain parameter has evolved over a long time interval.

Time Series Databases (TSDBs) have been developed in order to collect, persist and analyze these kinds of data efficiently. More specifically, the idea behind TSDBs is to

work with sequences of values, each with a time value indicating when the value was recorded. Their goal is to provide efficient ways to retrieve data form one or a few time series for a particular time range, often summarized or aggregated [25].

One of the challenges faced when dealing with time series data is that they are often generated at a very high rate, reaching millions of series datapoints quite frequently. Because of how event streaming platforms were designed (explained in section 2.1), they are very suitable for this scenario. For example, Kafka can be used to handle the ingestion of time series telemetry, that can later be persisted and processed by a TSDB [26].

Therefore, TSDBs can provide yet another way of querying data from streaming platforms, like streaming SQL engines, but focusing on retrieving timestamped data in an efficient way. Some of these platforms even provide a SQL interface to query time series data as if they were stored in a traditional relational database.

2.4.1 TimescaleDB

One of these systems is TimescaleDB. This database is implemented as an extension to PostgreSQL that introduces new capabilities for time series data management, functions for data analytics, new query planner and execution optimizations, and new storage mechanisms for more cost effective and performant queries [27].

It takes advantage of PostgreSQL's benefits, such as reliability, robustness, indexes, schemas, query planner, the SQL language, etc., and builds an efficient TSDB on top of all that [27].

Internally, TimescaleDB is structured around *hypertables*. While it may seem that a user is interacting with regular tables, under the hood it is a little more complex. Each hypertable is partitioned into one or more dimensions resulting in *chunks*. All hypertables are partitioned by the time column, and additionally they can be partitioned by other columns [27].

Each of these partitions, or chunks, are actually implemented as a regular PostgreSQL table. This is done to improve performance. First, working with smaller tables means that data and indexes for the most recent data are always kept in memory, so read and writes are much faster [27].

This partitioning design also allows to do something rather difficult for traditional relational databases, which is to distribute a hypertable across multiple nodes. By doing so, TimescaleDB can scale inserts and queries to adapt to the number of inserts and queries received at any time [27].

This design can be seen as a way to get the best of both worlds between wide and narrow databases; on the one hand, the good performance of wide structures on large datasets; on the other hand, the benefits of conventional indexes and the relational model. The user sees a wide structure, but the data is internally split.

TimescaleDB also has a similar concept to the materialized views we have seen, *continuous aggregates*, that incrementally maintain the materialization of aggregate data to return

results much faster. Even though this solution needs to perform refreshes on a established basis (unlike the ones provided by streaming SQL engines that responded to updates on the data source), it provides *real-time aggregation* to combine cached aggregates with the most recent data from a table, providing fresh results seamlessly for the user [27].

And one of the biggest advantages of this platform are the time optimizations. For example, as the time range for each chunk is known in advance, sorting can be optimized because only the data inside a chunk needs to be sorted [27].

2.4.2 InfluxDB

Another approach to time series databases is brought by InfluxDB. InfluxDB is a time series NoSQL database in which each data instance is structured as a timestamp, an associated set of tags that represent the metadata associated with a measurement, and a set of fields that are the actual values [28].

The main difference between tags and fields is that tags are always indexed, while fields cannot be indexed. Therefore, data that are commonly filtered by or grouped by are recommended to be specified as tags [29].

InfluxDB also provides a different query language. While TimescaleDB is full SQL compliant (with certain extensions), InfluxDB provides its own query language, *Flux*. It is a functional data scripting language that is quite far from the traditional SQL language [29].

As for the optimizations InfluxDB has taken, there are a few worth mentioning [29]:

- **Time-ordered data**. All the data written to InfluxDB is appended in time-ascending order, for performance purposes.
- **Eventual consistency**. InfluxDB prioritizes read and write requests over strong consistency, Any transactions that affect the queried data are processed subsequently to ensure that data is eventually consistent. Therefore, if the ingest rate is high (multiple writes per ms), query results may not include the most recent data.
- **Duplicate handling**. For performance and simplicity purposes, InfluxDB assumes that data received more than once is duplicate and it is updated with the most recent field value.

As it can be observed, InfluxDB is quite different from TimescaleDB in many aspects, even though in the end, they serve the same purpose.

In general, TSDBs are optimized for one specific scenario, the management of time series data. Nevertheless, with the growth of SQL streaming platforms, it is possible to perform these tasks with a different approach, even though until this moment, no streaming platform has really given too much importance to this field. Therefore, there is still room for adapting existent streaming solutions (or developing new ones) to support time series data with optimizations and extensions.

3 Development

This section is intended to present the design and implementation of the ChronoSQL project.

The following assumptions have been made to lay the foundation of the project:

- Event data is uninterpreted: the library does not know anything about what an event's payload bytes mean or represent
- Thus, queries will only use the ChronoTick (timestamp) of the events
- The ChronoTick representation is C++'s *std::time_t*
  - *std::time_t* is an integral value
  - It is implemented as a long int
  - It is usually used to hold the number of seconds since 00:00, Jan 1, 1970 (Unix time)
  - Therefore, the ChronoTicks used will have a granularity of seconds
- Event data will have a fixed size. This has been done for simplicity purposes, but in theory, non-fixed sizes could be implemented in the future relatively easily.

3.1 Initial goals

Initially, the development has been guided with the goal in mind of supporting a subset of queries that were interesting for the project. Therefore, all the development efforts have been put to implement and test the functionalities that would interpret the following five queries:

1. Simple selection (*GetAll*)

   The most basic query that can be written, and that we would need to support because it would also be the foundation of more complex queries, was a simple star selection. This query should simply return all the events in a Chronicle:

   $SELECT\ *\ FROM < log\_name >$

2. Time filtering (*GetByTime*)

   Our next goal was to start supporting some time-related functionality. And one of the most basic, yet most powerful, operator, would be the *WHERE* SQL operator, in conjunction with time comparison. For this, we designed the following query:

   $SELECT\ *\ FROM < log\_name >\ WHERE\ EID\ (<, >, \leq, \geq) < timestamp >$

   This query builds up from the basic selection and includes some inequalities operator to compare the Event ID with a timestamp.

3. Aggregation (*CountAll*)

   Next, we thought it would be interesting to give a way to count elements, and we included the *COUNT* aggregation operation:

   SELECT COUNT(*) FROM < log_name >

4. Filtering by day of the week (*GetWeekend*)

   This advanced filtering allows to select events that happened during a specific day of the week. For example, we can ask *"Find the events of every Monday in the last month"* to retrieve only events that happened on a Monday. The query is supported as follows:

   *SELECT * FROM < log_name > WHERE EID = 'day_of_the_week'*

5. Windowing: grouping by day and counting events (*CountByDay*)

   Finally, the last functionality we wanted to support were time windows. Researching timeseries databases, we observed that this was a very common functionality among them. Thus, we decided to add it to ChronoSQL. Queries with time windows can be expressed like:

   SELECT WINDOW(1 day) AS one_day, COUNT(*) FROM < log_name > GROUP BY one_day

   As it can be observed, it uses aliasing and grouping to allow to perform this operation. More details on the specifics of the implementation will be provided in Subsection 3.2.2.

3.2 High-level design

Figure 9 shows an overview of the high-level design of the project. There are two main components:

- **ChronoSQL** library: this is the actual ChronoSQL module. It receives SQL sentences and generates API calls to ChronoLog to fetch the corresponding data. It is also subdivided into two main components:
  - **SQL parser**, in charge of parsing, validating and generating query trees.
  - **Tree interpreter**, transforms the output from the parser into ChronoLog calls and performs any necessary processing on the data
- **ChronoLog** mock: a mocked version of ChronoLog that stores key-value events. It has three

A more in-detail explanation of this structure can be found in the following subsections.

3.2.1 SQL parsing

In order to read and validate SQL statements, we opted for using an external library. After exploring different alternatives, we chose to use an already existent library because

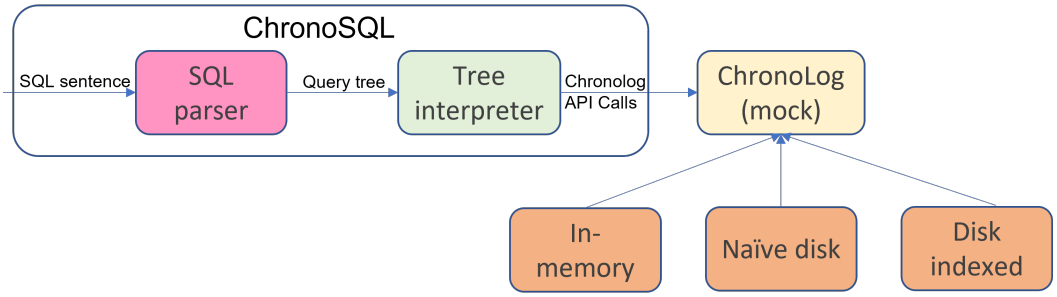

Figure 9: ChronoSQL high-level design

Features	sqltoast	usql	Hyrise parser	GSP	libpg_query
License	Apache-2.0	BSD-3-Clause	MIT	Proprietary	BSD-3-Clause
Last updated	2020	2017	Periodically	-	Periodically
Customizable	Supported	Supported	Very well supported and documented	-	Harder because it relies on the PostgreSQL parser
Pros	Small project size (13mb)	-	- Periodically updated - Easy customization - Small project size (7mb)	-	Full PostgreSQL syntax support
Cons	- No alias - Custom functions not recognized by default - Not maintained since 2018	Impossible to install (many errors)	Cumberstone to get all the info from a query	- Proprietary - Purchase required	- Developed for C - Harder to use - Intended as a base library for other languages - Overkill (it contains several components from PostgreSQL) Huge project size (136mb)

Table 2: SQL parsers feature comparison table.

developing one from scratch would require a high development effort, and we found several libraries that could parse SQL. This way, we could focus on implementing functionalities, rather than doing something that has already been done.

For this, we explored different libraries and compared them. The studied libraries are the following:

- sqltoast
- usql
- Hyrise parser
- GSP
- libpg_query

The following table summarizes the features and characteristics of each library:

With all this information, we decided to choose the Hyrise SQL parser mostly because it has wide SQL support, it is easy to extend with new functionality, and the library size is small.

This parser is used as follows:

1. A SQL statement is received by ChronoSQL
2. The statement is sent to the Hyrise SQL library which is in charge of:
   (a) Parsing
   (b) Validation
   (c) Query tree generation
3. The parser return a query tree that represents the initial statement

An example of how this parser works is the following. Let us consider this simple query:

*SELECT * FROM* < *log_name* > *WHERE EID* > 1628202739 *AND EID* < 1659738749*;*

With this query, the parser would generate a tree like Figure 10:

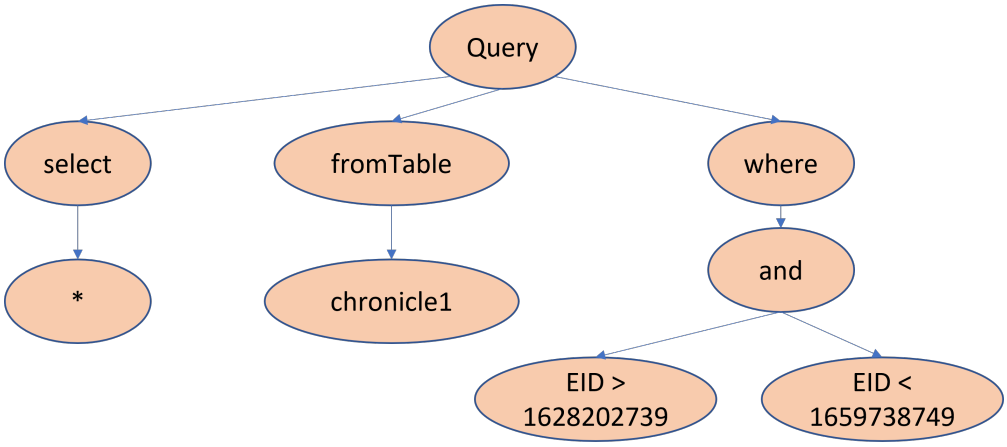

Figure 10: Example of a query tree

As it can be observed, it splits the query intro logical groups (select, from, where). If the query had more operators, such as a *GROUP BY*, they parser would generate another upper-level node. And as it can be observed, the tree has a recursive structure. If the query had included more AND/OR operators, more nested nodes would be present.

In order to give support to some time functionality (windowing and recognizing days of the week) the parser was modified. Internally, the Hyrise parser library was developed using C++ and a general-purpose parser generator named Bison.

Therefore, to include new functionality, first the Bison grammar was modified to support the new syntax rules. This meant adding a couple entries to the file, so that, for example, the window intervals were recognized as an integer plus a interval size (day, week, etc.).

Then, these new expressions were included in the *.h* C++ files of the project. Following the previous example, a new Interval Expression subclass had to be implemented to hold this data.

Finally, the API had to be modified so that the new pieces of information were returned when parsing a SQL statement.

3.2.2 Tree interpreter

This element is the primary component of ChronoSQL. It is the one that translates from a query tree to ChronoLog API calls that fetch the data, and performs any relevant computations on that data.

It identifies the different operators of the query tree:

- Selection
- Filtering
- Grouping
- Functions

Some of this information is used to fetch data from ChronoLog. For example, the *WHERE* clauses that specify a time range are used to fetch those events from ChronoLog that fall inside the specified time range. ChronoLog only receives a chronicle ID, and, optionally, the start and end EIDs that should be retrieved. The rest of the processing is handled by ChronoSQL.

- **Aggregation**.

  Aggregations are computed after the events have been retrieved from ChronoLog. For example, to compute a *COUNT* aggregation, the number of events in the returned list is obtained.

- **Windowing**.

  Windows group events depending on their timestamp. ChronoSQL supports six window sizes:

  - Second
  - Minute
  - Hour
  - Day
  - Month
  - Year

  After retrieving the corresponding events, the library loops through all of them while creating groups of events that fall inside the same window. The end of a window is recognized by the first event that falls outside the time range of that window, and a new one is created.

- **Days of the week**

  ChronoSQL can recognize days of the week. It is a useful functionality to filter events based on when they happened, for example, to extract the number of cars that entered a highway on the weekend vs the cars that entered during the rest of the week.

To compute this, first all the events that meet the other criteria are fetched. Then, the library goes through all the events and checks, for each event, if it happened in one of the days of the week provided with the query.

To extract the day of the week, the following is done:

1. Transform from seconds since 1970 to days since 1970:
   $days = floor(seconds/86400)$
2. As January 1st, 1970 (the Unix time starting point) was Thursday, add 4 to the obtained number of days: $days = days + 4$
3. Apply modulo 7 to get the actual day of the week: $week\_day = day\%7$

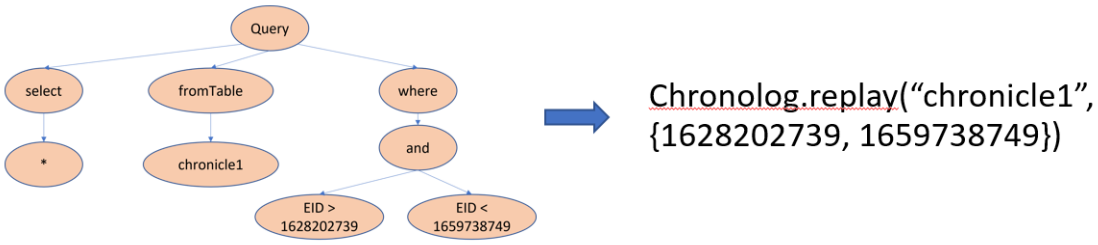

Figure 11: Example how a query tree is translated to an API call

Figure 11 illustrates the basic functioning of the tree parser component. From the tree generated in the example of the previous subsection, it interprets it and makes one API call to ChronoLog that obtains the events that were requested.

3.2.3 ChronoLog mock

As the ChronoLog project is still in the early development phase, we had to implement a *mock* of the platform to fetch data for ChronoSQL. In this mock, we represented the key-value events as:

- ***ChronoTick*** is represented as a std::time_t
- **Payload** represented as a *const char** (uninterpreted)

Initially, we created two different implementations of the ChronoLog API, an in-memory storage of events, and a naive disk storage. The first one stored all the events in a list in memory, and the second one persisted them to a *.log* file.

After developing them, we made some initial evaluations and saw that the performance gap between them was very big. As a result, we decided to investigate and implement some sort of indexing on the events so that data retrieval would be faster.

After exploring traditional solutions, such as B+-trees and LSM trees, we found the CR-index, a lightweight indexing solution for log-structured storage introduced by [5].

According to the design proposed by this paper, the log is logically split into groups of records of variable size that can be adjusted depending on the specific characteristics of the data being stored. Each index entry will hold the following information:

- Lower and upper boundary of the block (min and max values)
- Length of the block
- Location of the data pointed by this entry (file, offset)
- Pointer to the next entry

An overview of this indexing approach can be observed in Figure 12. Figure 13 shows how a query can be interpreted with this system, from the stored continuous data to blocks of data. It can be observed that certain blocks will meet the query's criteria, those that have at least one element matching the query requirements. Those blocks of data are then fetched from disk, and the elements are filtered as it would be done if no index existed, with the difference that the number of elements that are processed is less in most cases.

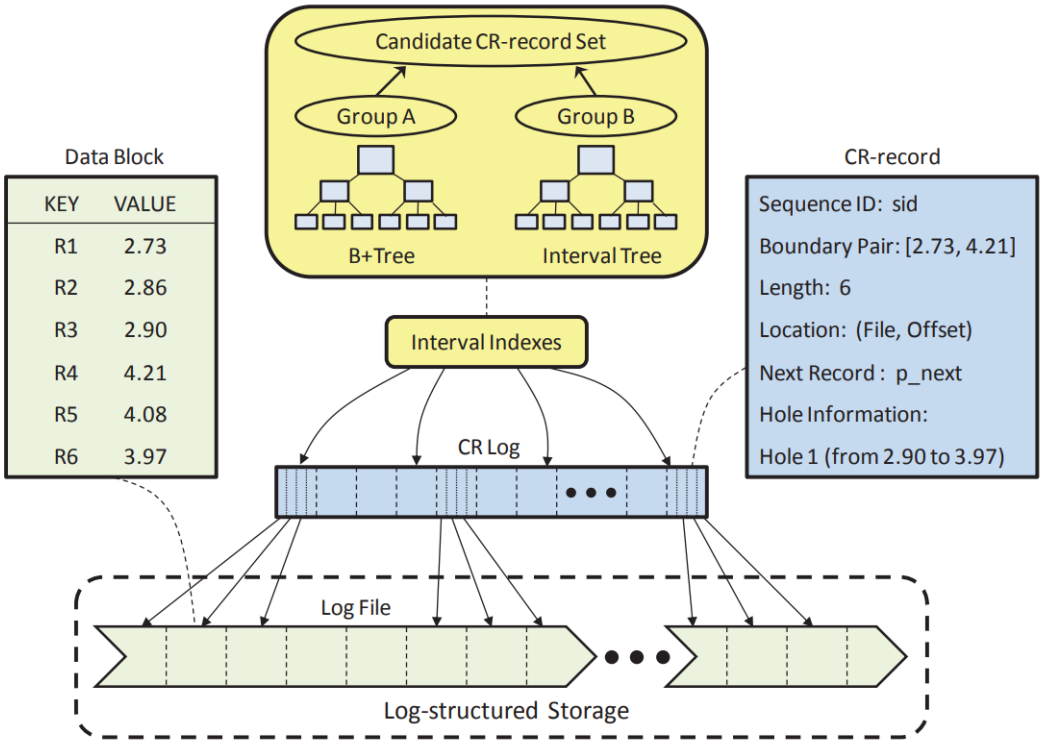

Figure 12: CR-indexing high-level structure [5]

This indexing has several benefits, according to the paper that proposed it [5]. In this paper, a comparison with two other common indexing techniques, B+-trees and LSM-trees, is conducted. The main comparison points are the following:

- The CR-index is very **lightweight**. Compared to the other approaches, its size is 10-12% of the LSM-tree, and 4-6% of the B+-tree.

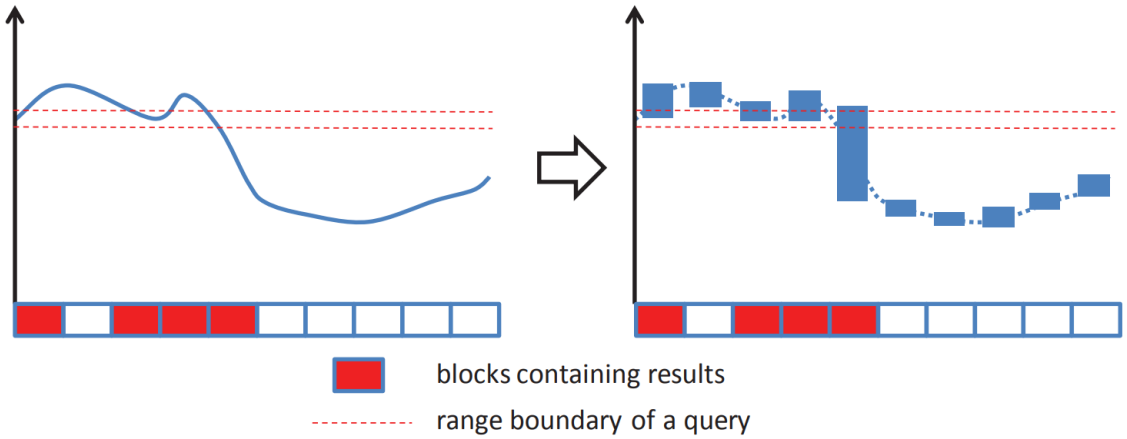

Figure 13: CR-indexing high-level structure [5]

- The CR-index has **low write overhead**, at most an increase in 8% (compared with 44-77% of an LSM-tree, and a 78-124% of a B+-tree). This is a key point for this system, because as ChronoLog is very focused on delivering very high write performance, a high overhead would diminish its benefits.
- **Low update time**, 15% of LSM-tree's update time, and 9% of B+-tree's.
- **Comparable response time**. The CR-index has a much lower lookup cost (3-10% of the compared approaches), mostly because it holds many less entries; on the other hand, its data access cost is higher, around 26-24%. At the end, this results in an overall response time that is quite similar.

For very large projects, one possibility the paper explores is to index the CR-index through a more traditional means, such as a B+-tree. In our case, we have decided it would be enough with one level of indexing, but it is an option that could be explored in the future if needed.

Additionally, one point that is not mentioned in the paper but we have found important is the simplicity of this design. CR-indexing can be achieved in a relative simple way because its structure is not very complex. On the other hand, the other two approaches require substantially more development effort.

Because of this, we can say that this indexing techniques is very suitable for our project, and it is the indexing approach we have chosen to apply. Figure 14 shows how we have designed the indexing for this project, making an adaptation of the CR-index [5] to our project. It is also very similar to the approach Kafka uses to index its events [30].

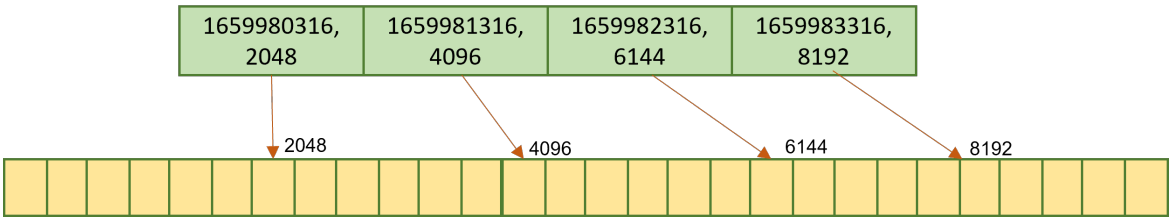

Figure 14: Implementation of the CR-indexing in ChronoSQL

As it can be observed, the index holds key-value pairs of a timestamp (the event ID), and an offset representing the position where the corresponding event is stored.

The way this indexing works is the following. Events are written to the *.log* file as it is done in the naive implementation. Once a byte threshold (configurable) is reached, an entry is appended to the index. In the diagram shown in Figure 14, this threshold would be 2048 bytes.

In order to retrieve an event using this technique, the index looks for the closest value that is less than the searched timestamp. In the example shown, if we looked for an event with ID 1659982356, the index would return the third index entry (1659982316, 6144) because that would be the closest value. With the offset obtained, the event is looked for starting at that position, instead of starting at the beginning of the log.

If a range of events is requested, the first timestamp of the range is searched in the index. Then, a scan is started from the returned offset until an event that is not contained in the range is found.

4 Evaluation

This section describes the evaluation that was conducted on the ChronoSQL project.

4.1 Dataset

In order to evaluate the project, we needed to generate a relevant set of events that would allow us to conduct the experiments. For this, we chose to randomly generate events that would span 365 days (from August 5th, 2021, to August 5th, 2022).

Even though the data is generated randomly (random ids, random payload contents), the generation has followed a uniform distribution on the event ids. This is, the generated timestamps are distributed evenly across the time range, so, for example, it is likely that the number of events in September 2021 is equal to the number of events in February 2020, and the number of events in the first half is equal to the number of events in the second half.

To evaluate performance, several sets were generated with different number of events:

- Set 1: 10,000 events
- Set 2: 100,000 events
- Set 3: 1,000,000 events
- Set 4: 10,000,000 events

4.2 Test environment

The evaluation was conducted in an environment with the following characteristics:

- **OS**:
  - Manjaro: version 5.10.133-1-MANJARO
  - Run in a Virtual Machine (VirtualBox 6.1.14 r140239)
- **Hardware**:
  - 8-core, 16-thread AMD Ryzen 4800HS CPU
  - 12GB RAM
  - 512GB PCIe 3.0 NVMe M.2 SSD
- **Software**:
  - C++ version 17
  - gcc version 12.1.0

4.3 Queries executed

The following are the queries that were executed in this evaluation (the first five are the ones part of the initial design shown in Section 3.1):

0. **(GetAll)** *SELECT * FROM < log_name >*
1. **(GetByTime)** *SELECT * FROM < log_name > WHERE EID >* 1628780806 *AND EID < 1658105202*
2. **(CountAll)** *SELECT COUNT(*) FROM < log_name >*
3. **(GetWeekend)** *SELECT * FROM < log_name > WHERE EID = 'Saturday' or EID = 'Sunday'*
4. **(CountByDay)** *SELECT WINDOW(1 day) AS one_day, COUNT(*) FROM < log_name > GROUP BY one_day*
5. **(CountByMonth)** *SELECT WINDOW(1 month) AS one_month, COUNT(*) FROM < log_name > GROUP BY one_month*
6. **(GetByTimeEnd)** *SELECT * FROM < log_name > WHERE EID > 1658105202*
7. **(CountByDayFiltered)** *SELECT WINDOW(1 day) AS one_day, COUNT(*) FROM < log_name > WHERE EID >* 1641936187 *AND EID <* 1652395282 *GROUP BY one_day*
8. **(CountMonday)** *SELECT WINDOW(1 day) AS one_day, COUNT(*) FROM < log_name > WHERE EID = 'Monday' AND EID >* 1641936187 *AND EID <* 1652395282 *GROUP BY one_day*

As it can be observed, the last four queries are a combination of the previous ones. Also, they are thought so that some conclusions can be extracted from the results of executing them. *GetByTimeEnd* is very similar to *GetByTime*, but it asks for events with an ID that is towards the end of the time frame used to generate events. Also, *CountByMonth* and *CountByDay* only differ in the window size.

4.4 Correctness evaluation

Before making the deployment and execution of the batch of performance tests, a correctness evaluation was conducted. Even though during the development of the library, periodical testing was done, all the queries that were part of the evaluation were first checked to ensure the results were correct.

For this, the 10k dataset was used, and each query was executed through the ChronoSQL terminal interpreter, and then the results were manually validated against the data.

For future improvements, automated tests could be developed to make the process more efficient and less fault prone.

	q0	q1	q2	q3	q4	q5	q6	q7	q8
10k	10,000	9,200	10,000	2,874	365	13	401	118	35
100k	100,000	92,869	100,000	28,509	365	13	5,201	121	35
1m	1,000,000	929,429	1,000,000	284,437	365	13	52,101	122	35
10m	10,000,000	9,298,637	10,000,000	2,848,746	365	13	517,501	122	35

Table 3: Number of results for the different queries and dataset sizes.

4.5 Number of results

Table 3 shows the number of results each query generates, for each dataset size. For most queries, the number of results increases linearly as we increase the number of events. Nevertheless, the following queries results do not scale as we scale the number of events:

- Query 4 (CountByDay): the number of results for this query is fixed because it is a windowing query, and the number of days in the year is fixed.
- Query 5 (CountByMonth): for the same reason, this query only shows 13 results, for the 13 months that the events span
- Query 7 (CountByDayFiltered): again, this is a windowing query. The small increase in number of days is probably because with a lower number of events it is less likely to have every day in the time range.
- Query 8 (CountMonday): as we are just selecting the mondays in the time range, and grouping by day, we always end up with a fixed amount of days.

4.6 Performance results

This subsection presents the results of executing the batch of queries with the different datasets and different ChronoLog implementations. All the code used in the evaluation (and the entire project) is available at [1].

The goal of the evaluation was to extract some insights on different aspects of the project. For this reason, all the queries were run against every dataset, obtaining the pertinent execution times, and several experiments were conducted. They are summarized in the following subsections:

4.6.1 Performance comparison as we increase the number of events

First, we wanted to know what was the performance degradation as we increased the number of events retrieved. This can be observed in Figures 15, 16, 17.

[1] https://github.com/scs-lab/ChronoSQL

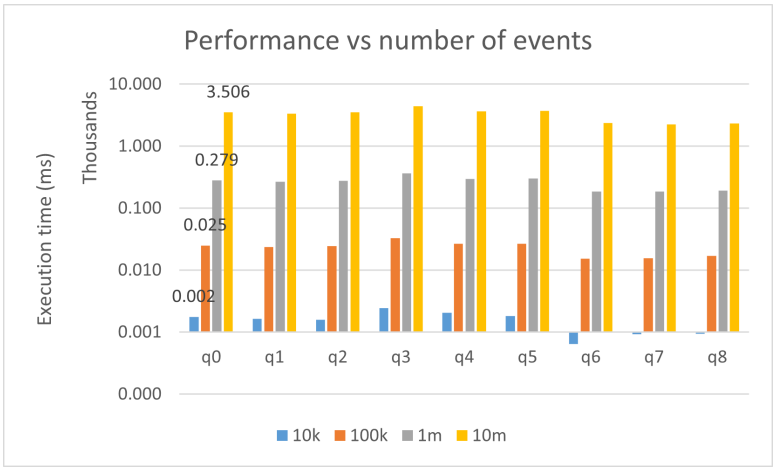

Figure 15: Execution time for the memory implementation (logarithmic Y axis)

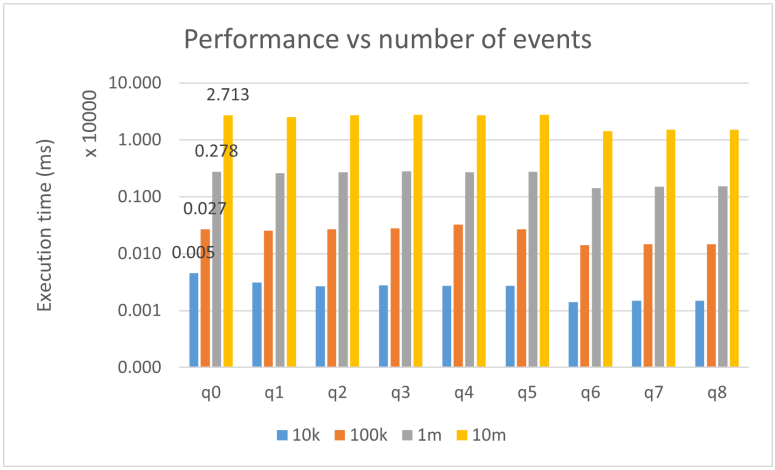

Figure 16: Execution time for the naive disk implementation (logarithmic Y axis)

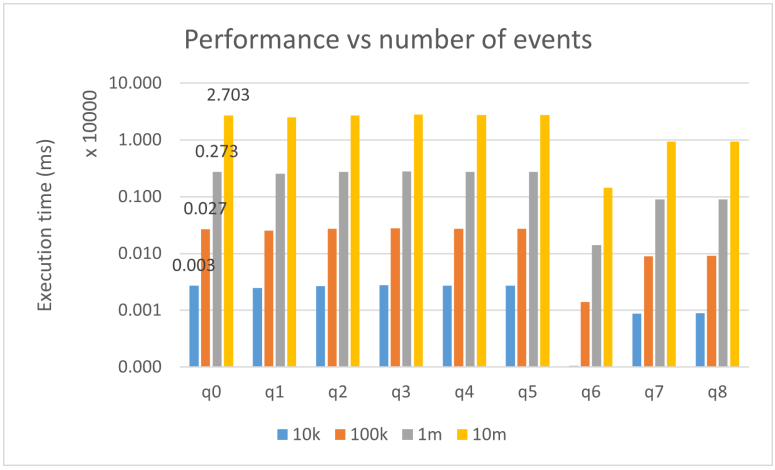

Figure 17: Execution time for the indexed disk implementation (logarithmic Y axis)

We can observe that the difference in execution times (considering the fact that the Y axis of the figures are in logarithmic scale) grows linearly with the number of events. This is, when we increase the number of events by ten, the execution time increases ten times as well.

Also from this data, we can see that the greatest part of the execution time is employed retrieving data. This can be extracted, for example, from the results of queries 0, 2, 3, and 4 (GetAll, CountAll, GetWeekend, CountByDay). These queries extract all the events, and the three later ones perform some additional computation on those events.

We can see from the results that there is almost no noticeable difference in execution time for the four queries, which probably means that the most costly operation is the data fetching, while the post-processing only represents a 5% overhead at most.

It is also interesting to see that there is not much performance difference between CountAll and CountByDay. This means that the windowing operator computation will not create a big overhead in the overall query execution.

The same can be said with the day of the week filtering. As the execution times for GetAll and GetWeekend are very similar, we can say that the day of the week filtering does not affect performance significantly.

4.6.2 Performance comparison among implementations

We also wanted to compare the performance of the three ChronoLog implementations. Figures 18, 19 show the execution times of the three implementations for 10k and 10m events, respectively (for 100k and 1m events, the results were very similar, so they are omitted for readability purposes).

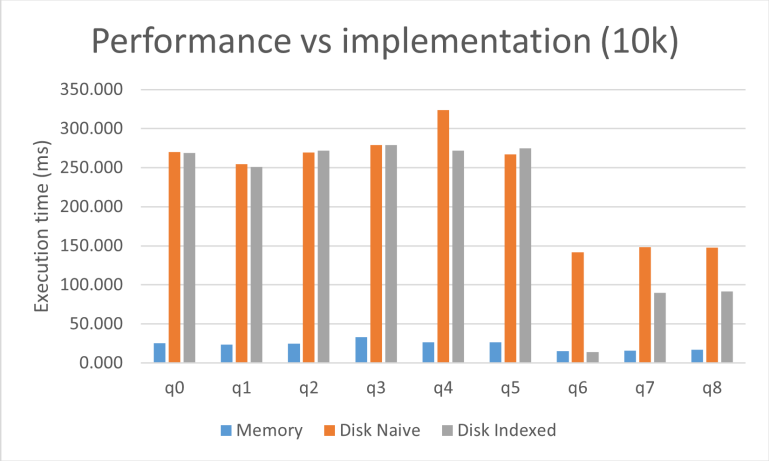

Figure 18: Execution time comparison for the three implementations (10k events)

It can be observed that memory clearly has a big performance gap compared to the other two implementations. This is probably to the overhead of disk I/O operations to fetch all the events that are required.

As for the differences between the other two implementations, we can see that for some queries (0 to 5) the difference is very small. Nevertheless, for the other queries (6 to 8)

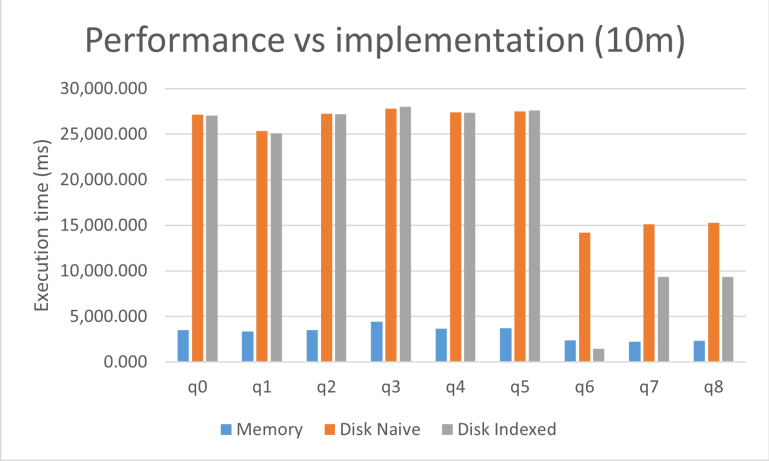

Figure 19: Execution time comparison for the three implementations (10m events)

the difference is very noticeable. The biggest difference occurs in query 6, which is even faster in the indexed implementation than in memory.

This performance gap has to do with the design of those specific queries. As queries 6 to 8 only require a small subset of the events, indexing is used to speed up the initial data fetching, avoiding to go through all the events to reach the start of the interval (this is why query 6 can be faster with indexing than in memory).

For the other queries, it is also a good indicator that there is no perceivable overhead on using indexing in queries that do not take advantage of it.

4.6.3 Payload size comparison

Finally, we wanted to evaluate how modifying the fixed payload size would affect the performance of the library. For this, we used the indexed implementation, and we executed the batch of queries modifying the payload size for each execution. The results of this experiment can be observed in Figure 20.

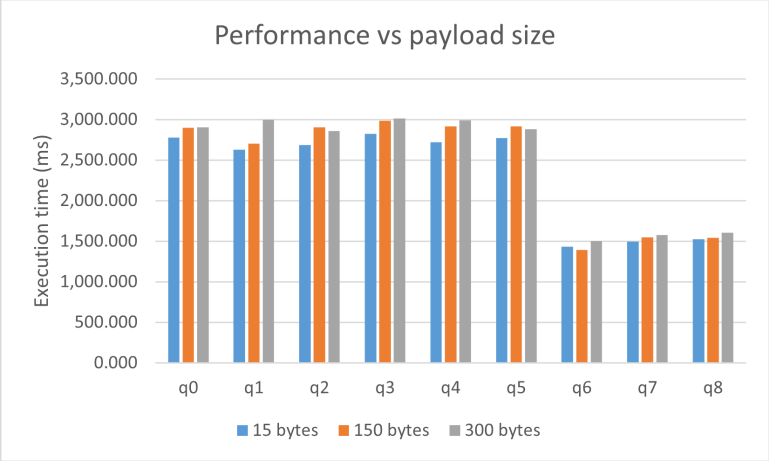

Figure 20: Execution time comparison for increasing payload size

As we can observe, the performance degradation is relatively small (at most 10%) when

we increase the payload size by 10 times, and the same when we double it from 150 bytes to 300 bytes. Therefore, we can deduct that working with larger event sizes should not make a difference compared with working with smaller sizes.

5 Conclusion and future work

In this report we have presented ChronoSQL, a SQL interpreter library for the ChronoLog platform. ChronoSQL presents a solution that allows to retrieve data from the ChronoLog system using a standardized programming language (SQL). ChronoSQL leverages the time-centered capabilities of ChronoLog to support enhanced and powerful querying capabilities on the system.

We presented several existing solutions from which we were able to extract design ideas and functionalities that helped us build this library. ChronoSQL stands apart from these other solutions by implementing time-related functions which are the core of the project and its biggest advantages. ChronoSQL can leverage log capabilities to provide powerful querying capabilities to streaming systems that are time-enabled, such as ChronoLog.

ChronoSQL presents both a SQL parser that transforms the received SQL query into a parsed tree of operations; a processing layer still on user space that transforms the query tree into the ChronoLog low-level API and operates on the received for filtering, aggregation and other such operations.

We also evaluated two different devices, memory and HDD, in which the storage layer of ChronoLog could be hosted. Significant differences were found between these two devices, as expected, and some work was executed to provide indexing capabilities to the slower layer (HDD) to improve its performance. With this indexing capabilities enhanced, our evaluation showed that the biggest computational efforts are used to retrieve data from the persistence layer, and that the computation overhead of the post processing ChronoSQL performs on the data is minimum.

Nevertheless, several future lines of work could be taken to improve this library:

- **Materialized views and streaming queries**: these are two of the biggest functionalities of existent streaming SQL systems. Nevertheless, implementing them would require not only a big development effort (outside the time scope of this project), but also more interactions with ChronoLog (to be notified when a new event arrives, for example) that as of today is not designed. Future implementations of the library could make use of these ChronoLog functionalities to offer them.
- **Distributed implementation**: for the moment, this implementation is ran on a single node that performs all the computation. A good enhancement would be to replicate these evaluations on distributed systems and explore enhancements under this conditions.

  A similar aproach to the Apache Presto architecture could be taken, where one master node is in charge of receiving requests and organizing the query execution, and several slave nodes are responsible for querying the data store and performing middle computations.
- **Wider SQL syntax support**: currently, only a small subset of the SQL operators are supported by the library. Implementing new operators (such as joins) is simple but time consuming, and is left to further development efforts in the library.

- **Query optimization**: with more SQL operations, query optimizations on top of the SQL tree that have been well studied on the literature could be useful to minimize the amount of events that are fetched from the persistance layer. And, a better interaction with ChronoLog could allow to further reduce the number of events to be processed by the library.

References

[1] M. Pathirage, J. Hyde, Y. Pan, and B. Plale, "Samzasql: Scalable fast data management with streaming sql," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1627–1636.

[2] Confluent, Inc., "ksqlDB Overview - ksqlDB Documentation." [Online]. Available: https://docs.ksqldb.io/en/latest/

[3] Materialize, "Home — Materialize Documentation." [Online]. Available: https://materialize.com/docs/

[4] The Presto Foundation, "Presto — Distributed SQL Query Engine for Big Data." [Online]. Available: https://prestodb.io/

[5] S. Wang, D. Maier, and B. C. Ooi, "Lightweight indexing of observational data in log-structured storage," *Proc. VLDB Endow.*, vol. 7, no. 7, p. 529–540, mar 2014. [Online]. Available: https://doi.org/10.14778/2732286.2732290

[6] A. Kougkas, H. Devarajan, K. Bateman, J. Cernuda, N. Rajesh, and X.-H. Sun, "Chronolog: A distributed shared tiered log store with time-based data ordering," 03 2021.

[7] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1789–1792.

[8] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1654–1655, aug 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824063

[9] Stack Overflow, "Stack Overflow Developer Survey 2021," 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021#overview

[10] J. Gama and M. M. Gaber, *Learning from Data Streams: Processing Techniques in Sensor Networks*, 1st ed., 2007.

[11] K. Me Me Thein, "Apache Kafka: Next Generation Distributed Messaging System," *International Journal of Scientific Engineering and Technology Research*, vol. 03, no. 47, pp. 9478–9483, 2014. [Online]. Available: http://ijsetr.com/uploads/436215IJSETR3636-621.pdf

[12] Amazon AWS, "What Is Amazon Kinesis Data Streams? - Amazon Kinesis Data Streams." [Online]. Available: https://docs.aws.amazon.com/streams/latest/dev/introduction.html

[13] VMware, Inc., "Messaging that just works — RabbitMQ." [Online]. Available: https://www.rabbitmq.com/

[14] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1634–1645, aug 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137770

[15] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 221–230. [Online]. Available: https://doi.org/10.1145/3183713.3190662

[16] D. Harris, "The team that created Kafka is leaving LinkedIn to build a company around it," 11 2014. [Online]. Available: https://gigaom.com/2014/11/06/the-team-that-created-kafka-is-leaving-linkedin-to-build-a-company-around-it/

[17] R. Chirkova and J. Yang, "Materialized views," *Foundations and Trends® in Databases*, vol. 4, no. 4, pp. 295–405, 2012. [Online]. Available: http://dx.doi.org/10.1561/1900000020

[18] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 1802–1813.

[19] M. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, 2018.

[20] "Differential Dataflow." [Online]. Available: https://timelydataflow.github.io/differential-dataflow/

[21] The Apache Software Foundation, "Samza - Documentation." [Online]. Available: https://samza.apache.org/learn/documentation/latest/

[22] ——, "ZooKeeper: Because Coordinating Distributed Systems is a Zoo," 03 2022. [Online]. Available: https://zookeeper.apache.org/doc/r3.8.0/index.html

[23] M. Thoma, "Library vs Service: 7 Arguments to consider - Level Up Coding," 01 2022. [Online]. Available: https://levelup.gitconnected.com/library-vs-service-7-arguments-to-consider-5088dced6022

[24] The Apache Software Foundation, "Samza - How to use Samza SQL." [Online]. Available: https://samza.apache.org/learn/tutorials/0.14/samza-sql.html

[25] T. Dunning and E. Friedman, *Time Series Databases: New Ways to Store and Access Data*, 1st ed. O'Reilly Media, 2014.

[26] A. Sargent, "Stream processing IoT time series data with Kafka & InfluxDB," 2021. [Online]. Available: https://www.confluent.io/events/kafka-summit-europe-2021/stream-processing-iot-time-series-data-with-kafka-and-influxdb/

[27] Timescale Inc., "TimescaleDB." [Online]. Available: https://docs.timescale.com/timescaledb/latest

[28] M. Freedman, "TimescaleDB vs. InfluxDB: Purpose-built for time-series data," 01 2022. [Online]. Available: https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/#:%7E:text=When%20it%20comes%20to%20data,NoSQL%2C%20non%2Drelational%20database.

[29] InfluxData, "InfluxDB OSS 2.2 Documentation." [Online]. Available: https://docs.influxdata.com/influxdb/v2.2/

[30] P. Patierno, "Deep dive into Apache Kafka storage internals: segments, rolling and retention," 12 2021. [Online]. Available: https://strimzi.io/blog/2021/12/17/kafka-segment-retention/

6 Annex

...