TSAggCache: An Efficient Cache Service to Support Repeated Queries on Time Series Data

Mingyi Lim
New York University
ml9027@nyu.edu

Eugene Chang
New York University
ec4338@nyu.edu

ABSTRACT

Time-series databases are the backbone to modern monitoring systems. While time-series databases are typically optimized for fast reads, the increasing number of users of these databases and the need to analyze large amounts of correlated time series prompts the need for an efficient solution to reduce load on time series databases. We present TSAggCache, a platform-agnostic, in-memory caching solution which exploits the similarities in time-series queries, as well as the properties of aggregated time-series data to improve read performance of time-series queries and reduce the load on time-series databases. We have developed a translation DSL, efficient cache indexing structure, and layout which allows the re-use of cached data over multiple dimensions. Our experiments show that TSAggCache can significantly improve client query performance, reducing latency by up to 91% while providing correctness guarantees.

1. INTRODUCTION

The importance of Time series data is growing immensely, with Twitter's Observability stack collecting 170 million metrics every minute and serving 200 million queries per day [1]. Time series data is used across a wide range of industries due to its ability to capture and record events over time, offering a dynamic view of change and progression. This type of data is essential for analyzing trends, understanding past behavior, and forecasting future occurrences, making it invaluable for strategic planning and operational efficiency. In fields such as economics, finance, healthcare, and engineering, time series data helps stakeholders identify patterns that would otherwise go unnoticed in irregular or cross-sectional data sets [8]. The advancement of IoT and smart devices has expanded the scope and scale of time series data collection. Real-time data streaming from these devices facilitates immediate analysis and reaction, enhancing the ability to manage resources efficiently and optimize operations [6]. In our increasingly data-driven world, the role of time series data is pivotal in driving innovation and efficiency.

However, many applications manage time series data by frequently re-querying, either repeatedly or upon startup. Such recurrent queries can overburden the time series database as they involve querying and aggregating the same data repeatedly, which is often unnecessary. In the context of time series data, this practice can significantly impact performance and scalability. While time series databases are optimized for handling large volumes of sequentially recorded data efficiently, when they are subjected to unnecessary load through constant re-queries, it can lead to slower response times and increased processing costs. Implementing caching strategies can help alleviate these issues, ensuring that the database maintains high efficiency and responsiveness.

2. BACKGROUND

2.1 Time Series Data

Time series data consists of observations recorded sequentially over time. This type of data is characterized by its chronological order, which is crucial for analysis, as the time interval between data points can impact the interpretation of trends and patterns. Commonly found in various fields such as economics, finance, healthcare, and engineering, time series data is often used to forecast future events based on historical patterns. For example, stock market prices, daily temperature readings, and continuous health monitoring data are all examples of time series. Analyzing time series data enables the identification of seasonal variations, cycles, and trends, providing valuable insights that aid in decision-making and predictive analytics [6].

2.2 Time Series Database (TSDB)

Time Series Databases (TSDB) have become increasingly popular in recent years and have been created to serve various purposes. A TSDB is a type of database optimized for storing and retrieving time-series data. Time series data is tracked, monitored, down sampled, and aggregated over time. This could include a wide variety of data types such as financial transactions, IoT sensor data, system performance metrics, stock prices, weather data, etc [5].

2.3 InfluxDB

Among the various time-series databases, InfluxDB [2] is one of the most highly regarded and widely used. InfluxDB is known for its effective storage, analysis, and retrieval of data metrics. It is often used for applications such as real-time analytics, network monitoring, and application performance monitoring [3].

InfluxDB is an open-source database dedicated to time-series data and is entirely developed in Go without any external dependencies. It processes SQL-like queries through InfluxQL, which can be executed via HTTP or JSON over UDP. While InfluxDB supports basic data aggregation, it does not allow for complex join operations. Its storage engine is built around a data structure known as the TSM-Tree, like the LSM-Tree, which enhances the efficiency of insert operations. Deletions are not directly supported in InfluxDB; instead, it automatically purges outdated data based on a predefined retention policy [4, 8].

InfluxDB organizes time-series data into shards, with each shard representing a discrete time range of data points and operating as an autonomous data management entity. Each shard possesses its own cache, Write Ahead Log (WAL), and TSM files. To ensure data reliability, incoming data is initially written to the WAL before being saved to the TSM files on disk. Each TSM file, a read-only file, stores compressed time-series data in a column format and is structured similarly to an SSTable in LevelDB [14], containing sections for the header, blocks, index, and footer. Shards covering non-overlapping time ranges can be grouped into shard groups that share a common retention policy, with expired shards being removed to free up resources [4].

InfluxDB operates without a fixed schema, allowing users to dynamically add measurements, tag sets, and field sets as needed. It treats measurements similarly to tables in traditional SQL databases, with each data point marked by a timestamp and capable of having multiple field and tag keys. Data points are stored in chronological order within the measurements [3].

2.4 InfluxQL

InfluxQL is a custom query language used by InfluxDB, designed to handle the specifics of time-series data through a syntax similar to SQL. This language enables users to perform various operations such as data insertion, deletion, and manipulation directly through HTTP or JSON over UDP. InfluxQL excels at straightforward data retrieval and aggregation tasks, allowing users to easily sum, count, average, and perform other basic aggregative functions across time-series datasets. However, it lacks the capability for more complex join operations, limiting its use in scenarios requiring extensive relational data analysis [9]. Despite this, InfluxQL remains a powerful tool for developers and analysts working within the InfluxDB environment, providing efficient and intuitive access to time-series data management.

2.5 Materialized Views

A materialized view is typically a data table created by combining data from multiple existing tables [10] and performing certain operations on them. This allows data re-use from the stored table when certain conditions are met. Materialized views are used in databases such as SageDB [11] for efficient retrieval of subsets of data.

2.6 Related Work

Previous work on caching time series data focused on time-slice based reuse of data [12, 13]. TSCache, a time-series cache system design by Liu et al., focused on the performance of the cache itself, and the efficient retrieval of data using memory management features such as slab caches and compaction processes. However, they do not focus on re-use of cached data in multiple dimensions such as tags, measurements, and aggregation windows. Our work aims to utilize these properties of aggregated data to find a logical way to structure and retrieve cached data.

3. DESIGN

In this paper, we present an efficient, platform agnostic time-series cache system design, called TSAggCache, to optimize client queries to time-series databases. The goal of TSAggCache is to provide a reliable cache service for clients using real-time time series data.

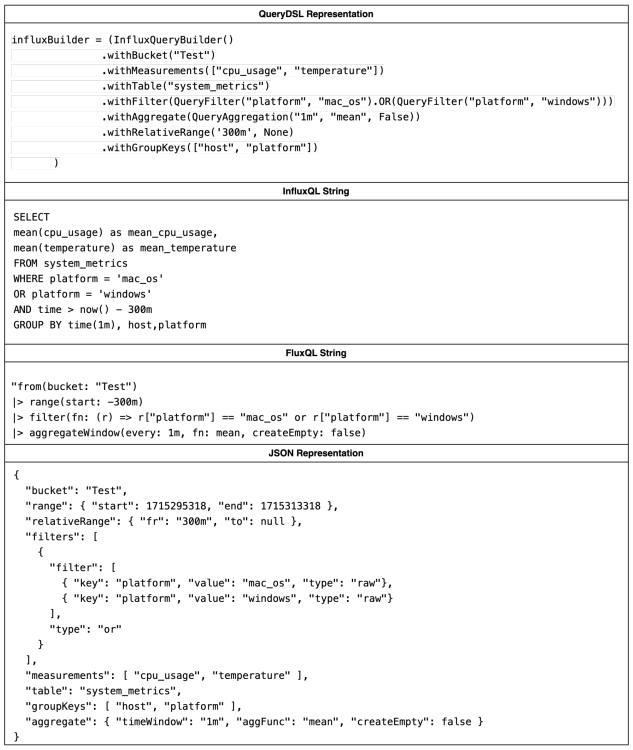
3.1 Query DSL

TSAggCache utilizes an internal DSL to build and process queries. This allows us to extend the functionality of

TSAggCache, allowing it to be used as a cache for other Time-Series database implementations such as Prometheus and Timescale DB in future. We achieve this by providing an extensible DSL which provides abstractions for core features of time series queries such as measurements, filters, tags, and aggregation functions. This allows us to translate queries between different languages and serialization formats, such as from FluxQL to InfluxQL and to JSON. This allows us to build queries using a unified DSL for multiple platforms.

3.2 Architecture

To allow multiple clients to reap the benefits of a shared cache, we designed TSAggCache to be deployed as an external service, which allows for horizontal scaling. This allows the service to re-use results queried from multiple clients to service requests. The cache service intercepts requests to the underlying database, checking if any re-usable components are present in the cache, before making a subsequent, smaller query to the underlying database.



QueryDSL Representation

```
influxBuilder = (InfluxQueryBuilder()
                .withBucket("Test")
                .withMeasurements(["cpu_usage", "temperature"])
                .withTable("system_metrics")
                .withFilter(QueryFilter("platform", "mac_os").OR(QueryFilter("platform", "windows")))
                .withAggregate(QueryAggregation("1m", "mean", False))
                .withRelativeRange('300m', None)
                .withGroupKeys(["host", "platform"])
        )
```

InfluxQL String

```
SELECT
mean(cpu_usage) as mean_cpu_usage,
mean(temperature) as mean_temperature
FROM system_metrics
WHERE platform = 'mac_os'
OR platform = 'windows'
AND time > now() - 300m
GROUP BY time(1m), host,platform
```

FluxQL String

```
"from(bucket: "Test")
|> range(start: -300m)
|> filter(fn: (r) => r["platform"] == "mac_os" or r["platform"] == "windows")
|> aggregateWindow(every: 1m, fn: mean, createEmpty: false)
```

JSON Representation

```
{
  "bucket": "Test",
  "range": { "start": 1715295318, "end": 1715313318 },
  "relativeRange": { "fr": "300m", "to": null },
  "filters": [
    {
      "filter": [
        { "key": "platform", "value": "mac_os", "type": "raw"},
        { "key": "platform", "value": "windows", "type": "raw"}
      ],
      "type": "or"
    }
  ],
  "measurements": [ "cpu_usage", "temperature" ],
  "table": "system_metrics",
  "groupKeys": [ "host", "platform" ],
  "aggregate": { "timeWindow": "1m", "aggFunc": "mean", "createEmpty": false }
}
```

Figure 1: DSL Translations to different languages

To minimize memory usage, TSAggCache only stores pre-aggregated results from time series queries in memory. This allows the cache to avoid storing individual datapoints, instead storing aggregations over periods of time, minimizing the amount of data stored by orders of magnitude. Storing aggregated data also allows for the efficient querying of data. Since data is aggregated over uniform periods of time, the cache is able to efficiently calculate and look up the specific data range required in *O(1)* time, without needing to perform a scan of all datapoints which would typically be an *O(n)* operation, or a *O(log(n))* operation with the uses of indexes or tree-like structures.

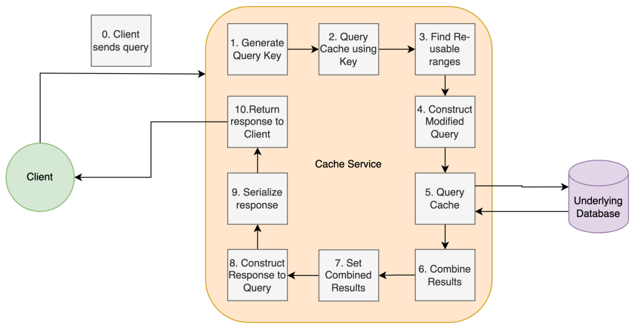


Figure 2: Architecture and request flow

TSAggCache utilizes a universal caching scheme which is independent of the underlying database used. Queries to the cache service are made by submitting a DSL object which represents the query. The cache service constructs the cache key from the DSL object, which is a combination of a database, table, and the aggregation function. TSAggCache then uses an inverted index to find entries which match the query based on measurements, filters, and tags requested. If a cache match is found, TSAggCache finds the portions of the matching data which contain re-usable components relevant to the query and constructs a new DSL object from the original query after re-use of the cached data. The cache then submits the minimized query to the underlying time-series database to retrieve the remaining data not found in the database.

Once the remaining data is retrieved, TSAggCache combines and sets the new data in the cache. It then constructs a response for the query using the cached data through a combination of operators and returns the constructed result to the client.

3.3 Query Modelling

Due to the inherent complexity of queries, we model queries in the following way to reduce the dimensionality and variation of queries. A query to the cache can be modelled as the following:

1. Hard Filters
    a. Table

   b. Query Filters
   c. Aggregation Function
2. Soft filters
   a. Measurements
   b. Tag Filters
3. Aggregations
   a. Aggregation Window
   b. Groupings
4. Time Ranges

We posit that most time-series queries can be broken down into the above components.

Hard filters represent the boundaries of what the cache can or cannot re-use. For example, queries for data from a specific table cannot be served by any cached data from any table. Likewise, data aggregated over a window from one aggregation function cannot be used to re-compute data for a different function due to the inherent information loss aggregations entail. Thus, the fields from the hard filters are used to construct the query key.

Soft filters are somewhat malleable. With a correct caching scheme and under the correct conditions, we can potentially re-use stored data in the cache to meet these criteria. For example, a query made with a set of measurements $\{a, b\}$ can be partially served by a cache which has stored data for measurement $\{b, c\}$ in the same table by re-using data in $\{b\}$. Likewise, a query requesting data for measurement *a* with tags (*tag1=x AND tag2=y*) can be served by a cache which has stored data for (*tag1=x)* by omitting data from the cached data with (*tag2 != y*). However, the converse does not hold. A query for data with (*tag1=x)* cannot be entirely served by a cache entry which has been filtered for (*tag1=x AND tag2=y*), since it contains an aggregation over less data than required by the query.

Aggregations from queries can always be served by the cache if the cached data is of higher granularity than the query. A query with request window of 30m can be served by combining previously cached entries with aggregation windows of 10m if the aggregation function is commutative. Similarly, data stored in the cache grouped by fields (a, b, c) can be recombined to serve a query which requests data grouped by (a, b) by re-partitioning entries based on the less-granular grouping.

Lastly, time-ranges are always reusable by the cache if the above criteria are met. If a query is submitted for a series stored in the cache, for a specific time range {t1, t2}, we will always be able to re-use any cached data within the time range {t1, t2} to reduce the size of the query.

4. IMPLEMENTATION

We construct the cache as a Key-Value store, with each value stored as a tree. Each value is indexed by the constructed key using the hard filters based on a combination of the table, aggregation function and hard filters.

Each value in the cache is represented by a Series object, which stores metadata about the key fields, as well as the aggregation interval, group keys, and range of data stored in the cache. Each series contains a set of SeriesGroup objects, where each SeriesGroup contains values with the same value for each tag. For example, given 2 tags, '*host*' and '*platform'*, each with 2 possible values {'*host1*', '*host2*'} and {'*platform1*', '*platform2*'}, we will construct 4 series groups, each holding data which contain tag values

1. {*host*='*host1*', *platform*='*platform1*'}
2. {*host*='*host1*', *platform*='*platform2*'}
3. {*host*='*host2*', *platform*='*platform1*'}
4. {*host*='*host2*', *platform*='*platform2*'}

Above is a representation of how time series databases store values with differing tags.

Each SeriesGroup contains a list of contiguous data points, uniformly spaced out in periods corresponding to the aggregation window used to retrieve the data. Lastly, each Series contains an inverted index to quickly search and filter for series which contain relevant data to a query.

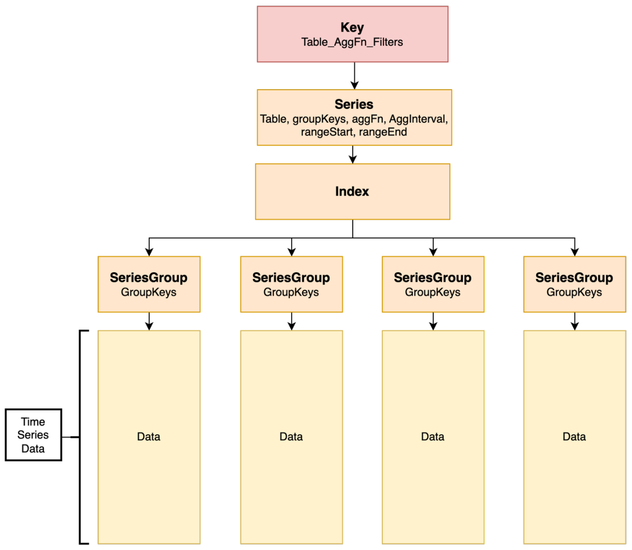


Figure 3: Implementation of a cache for a single key

4.1 Cache Search

When a query is submitted to the cache, the cache first constructs the query key, and checks if any data exists in the cache. If the cache contains a matching key, it then checks the aggregation windows and group keys to ensure that the cached data is of higher granularity than the query. Once the granularity of the data is verified, the cache service then compares the range of the query against the range stored in the cache to determine the minimized range required to be submitted to the cache.

4.2 Query Modification

The cache service modifies the incoming query as follows:

1. If the aggregation window of the query is larger than the cached aggregation window, the query's cached window is modified to match the aggregation window of the cached data. This is done to preserve the uniformity of the cached data.
2. If the group keys of the query are a subset of the cached group keys, we expand the set of group keys to match the existing group keys in the cache. Again, this allows us to preserve the structure of the cached data.
3. Lastly, we modify the range of query to the minimized range required to fulfil the request after incorporating the cached data.

The first 2 modifications above preserve the structure of the cache and ensure that the cached data is reusable. The last modification allows us to re-use the cached data to serve the query.

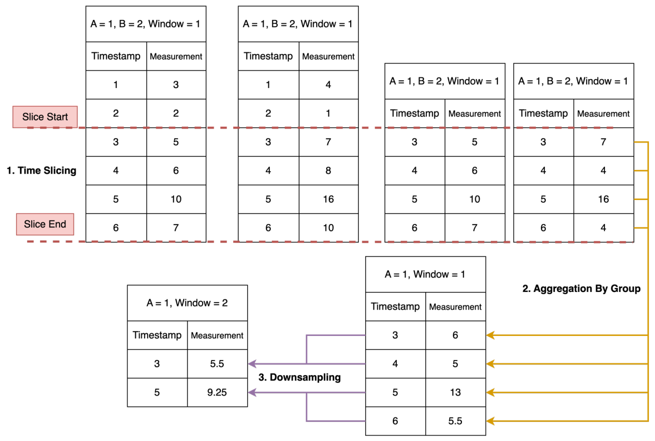
4.3 Querying the Database

TSAggCache then submits the modified query to the underlying database using a provided client. It then appends the results of the query to the existing Series, after re-grouping the results of the query. If no cache match was found, the original query is submitted since no modifications were made to the query, and a new Series object is created in the cache using the constructed key.

4.4 Reconstruction

The cache re-constructs the required results from the original query using a series of operations.

1. Slicing: The cache slices the data in the cache by efficiently looking up the start and end index of the original query. This is done by computing the required offsets in the array from the start of the stored range.
2. The sliced data is regrouped according to the groupings required by the client. Since the cache is guaranteed to store groups with equal or higher granularity than the query, this is done by placing SeriesGroups into buckets which match the client groupings. Then, data in each Series Group is combined at each index using the (commutative) aggregation function provided.
3. Once regrouped, the data is re-aggregated based on the required aggregation window. We use a down sampling function to re-aggregate the data and apply the aggregation function in the query.
4. Lastly, we filter the set of measurements required by the query.
5. Once done, we combine the results obtained from each SeriesGroup into a single table and return it to the client.



A = 1, B = 2, Window = 1	
Timestamp	Measurement
1	3
2	2
3	5
4	6
5	10
6	7

A = 1, B = 2, Window = 1	
Timestamp	Measurement
1	4
2	1
3	7
4	8
5	16
6	10

A = 1, B = 2, Window = 1	
Timestamp	Measurement
3	5
4	6
5	10
6	7

A = 1, B = 2, Window = 1	
Timestamp	Measurement
3	7
4	4
5	16
6	4

A = 1, Window = 1	
Timestamp	Measurement
3	6
4	5
5	13
6	5.5

A = 1, Window = 2	
Timestamp	Measurement
3	5.5
5	9.25

Figure 4: Example of the reconstruction process

4.5 Limitations

Due to the high network latency incurred in each request to the database, we place limitations on the results we can construct using this method. If any limitation is violated, we instead perform a full query to the database and construct a new result set.

1. If a query to the cache requires the cache to make multiple requests to the database due to range extensions, we simply query the database to obtain the entire range.
2. Non-Commutative functions. Non-commutative functions (such as medians) disallow us from re-aggregating and combining results unless
   a. The query's aggregation window matches the cached Series' aggregation window.

b. The query's group keys match the cached Series' group keys exactly.

4.6 Server Implementation

The cache service is implemented using multiple layers for modularity. The server layer is responsible for accepting requests and responding to the client. It is also responsible for serialization and deserialization of requests. The cache service implements all the logic for searching, query modification and reconstruction of data. Lastly, the storage layer is solely responsible for storing data in the format mentioned. This is shown in Figure 4.

We implemented the server layer using a Python Flask server, which provides a REST API for as an interface for clients. The interface provides 3 primary endpoints a GET /health endpoint for checking the status of the service, a POST /query endpoint to submit queries to and respond to clients and a POST /reset endpoint to clear existing cache entries which is primarily used for tests. Both the /health and /reset endpoints return a generic response to the client indicating the success of the operation.

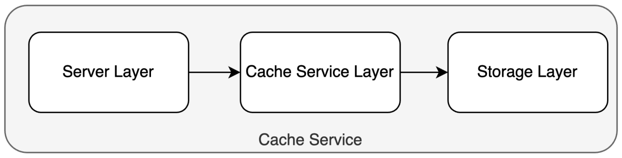


Figure 5 Structure of Cache Server

The /query endpoint is used as the primary interface to accept queries from clients. Clients can use a client library to communicate with the cache service, which translates the query to the cache DSL, or directly send a JSON object which corresponds to the JSON format of the request string. It then invokes the cache service layer for query processing.

5. BENCHMARKS AND RESULTS

We constructed a simple benchmark to test the latency gains using TSAggCache against direct queries to InfluxDB. The benchmark consists of 7 consecutive queries directed at a single table. The benchmark is designed to replicate the queries real-world users might use to fetch data from a time series database.

Query 1: Base Query	`SELECT mean(cpu_usage) as mean_cpu_usage, mean(temperature) as mean_temperature FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 300m GROUP BY time(1m), host,platform`
Query 2: Re-use measurements	`SELECT mean(cpu_usage) as mean_cpu_usage, mean(memory_usage) as mean_memory_usage FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 300m GROUP BY time(1m), host,platform`
Query 3: Extend Range	`SELECT mean(cpu_usage) as mean_cpu_usage, mean(temperature) as mean_temperature FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 600m GROUP BY time(1m), host,platform`
Query 4: Change Groups	`SELECT mean(cpu_usage) as mean_cpu_usage, mean(memory_usage) as mean_memory_usage FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 600m GROUP BY time(1m), platform`
Query 5: Downsampling	`SELECT mean(cpu_usage) as mean_cpu_usage FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 600m GROUP BY time(2m), host,platform`
Query 6: Downsampling with Group Change	`SELECT mean(cpu_usage) as mean_cpu_usage, mean(temperature) as mean_temperature FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 600m GROUP BY time(3m), platform`
Query 7: Downsampling with Group Change and Measurement Change	`SELECT mean(cpu_usage) as mean_cpu_usage, mean(temperature) as mean_temperature, mean(memory_usage) as mean_memory_usage FROM system_metrics WHERE platform = 'mac_os' OR platform = 'windows' AND time > now() - 600m GROUP BY time(10m), platform`

5.1 Data Generation

We deployed an instance of InfluxDB serverless on the cloud, and a data producer which continuously writes data from a server consisting of (temperature, CPU usage and memory usage) to a system metrics table at irregular intervals. An example of the datapoints produced is as follows:

Time	Platform	Host	CPU usage	Temperature	Memory usage
12412321	Windows	Host1	32.1312	37.2123	10.1232
13214219	Mac_OS	Host2	32.1315	38.1425	5.9421

5.2 Correctness

To first show that our cache implementation outputs the correct results, we compared the results over one run of the benchmark to ensure that the results we obtained were near identical. We created 2 different clients – The first sending the query to InfluxDB directly and the second doing so via TSAggCache. We ran the benchmark queries once each for both clients and recorded the results in a plot.

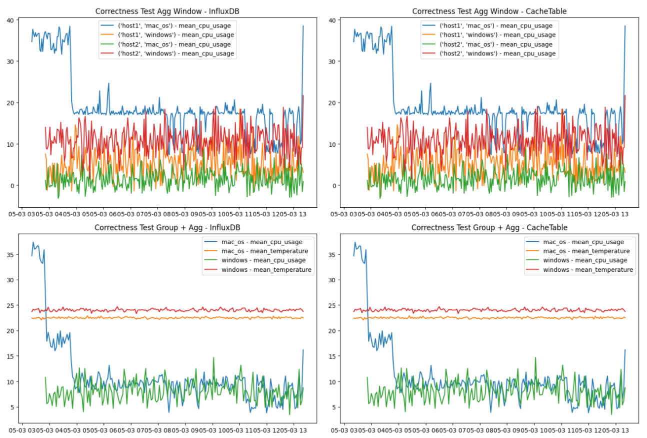


Figure 6: Results from the correctness test show that data from cache and InfluxDB are nearly identical

5.3 Latency

To assess the improvements in latency from re-use of components, we ran the benchmark using 2 different clients again. This time, we ran the benchmark for 100 iterations, resetting the cache after each iteration. We took the median (p50) as the average score to account for unpredictable spikes in latency from InfluxDB, which occasionally takes up to 20s to return.

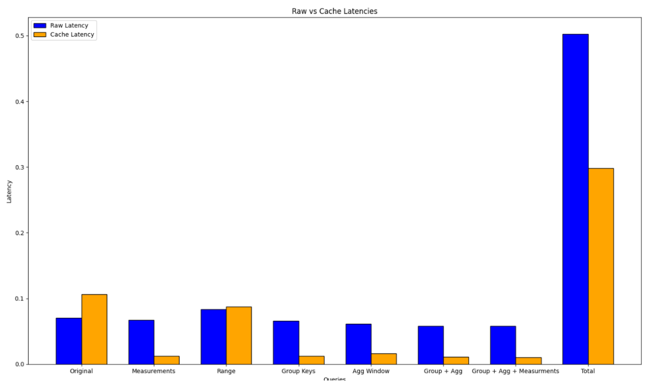


Figure 7: Latency of each query in the benchmark

Through the experiment, we observed a 40% reduction in latency via the cache service. While the original query took longer due to the additional overhead required for processing the query in the cache service, the gains in efficiency are seen in subsequent queries with an up to 91% reduction in latency in some cases where no additional query is required to the database. Interestingly, we observe the extended range query took a longer than the original query despite the re-use of previously stored ranges. This could be caused by the overhead in cache processing outweighing the efficiency gains in querying a smaller subset of the data from Influx.

5.4 Scaling

To observe the impact of scaling queries, we conducted another experiment to find the impact of increasing query sizes on the cache, while keeping the incremental constant. We used the same 2 clients again, and each client made 2 queries, one with the original range, and another with a slightly extended range. This allows TSAggCache to only fetch the incremental data required in the second range and allows us to find a point where the incremental range leads to performance gains. We then measured the p50 latency of the second query for both clients. In this test, we also included tracing of each step in TSAggCache to find bottlenecks in cache performance. We observe that as the initial query size increases, the relative gains from using pre-cached range data increases. While the latency of the InfluxQL query generally increases as the query size increases, latency of queries to TSAggCache remain relatively stable. Between queries for 360m and 720m, we observe a spike in latency which could be caused by InfluxDB retrieving multiple blocks to fetch the required data. However, since TSAggCache only needs to retrieve the latest 10m, it can more efficiently use InfluxDB. Within the queries to TSAggCache, we also see that query latency is still dominated by queries to InfluxDB despite the smaller range queries. This is an area of potential optimization for future uses.

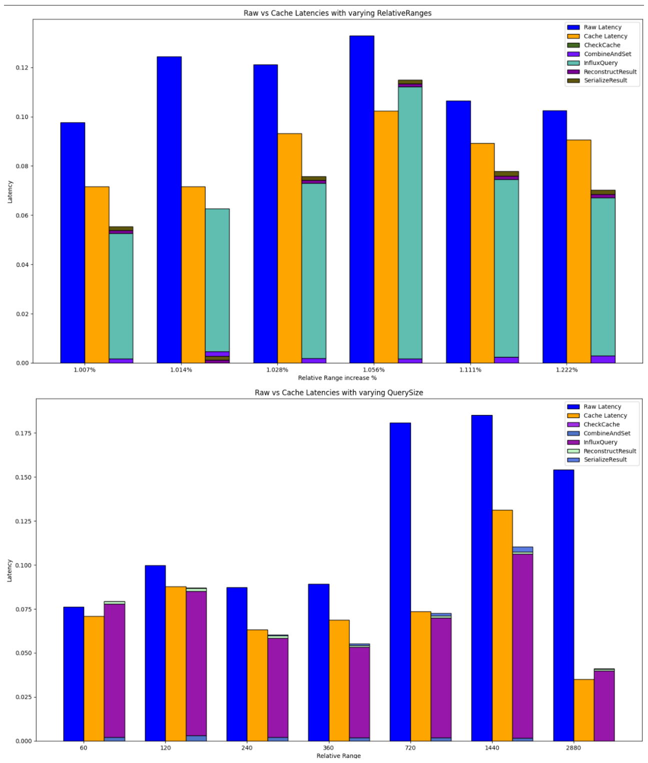


Figure 8: Breakdown of Latency with increasing range and increasing size.

6. CONCLUSION

In this paper, we presented TSAggCache, a proof-of-concept time series cache system design to optimize queries to InfluxDB and potentially other TSDBs. We have designed a cache service with a set of optimizations to improve the retrieval of time series data, and our results have shown the effectiveness of our proposed design.

7. FUTURE WORK

In future work, we plan to enhance the cache design by enabling it to accept writes as part of a write-through service. During our tests, the most significant source of cache latency arose from the additional queries needed to access InfluxDB. By adopting a write-through approach, we can preemptively insert relevant datapoints into the cache as they are written to InfluxDB, thereby eliminating the need for these extra queries. This change could significantly reduce latency and improve overall system efficiency. Additionally, we aim to increase the dimensionality of the cache. Currently, the high latency involved in querying InfluxDB prevents us from using cached data to return results that vary across multiple dimensions. Enhancing the cache's capability to handle more complex data dimensions will allow for faster and more flexible data retrieval.

Another area for improvement involves the implementation of filters. As of now, the cache does not support complex filtering due to its lack of awareness of the full schema of the underlying data. By transitioning to a schema-aware cache, we could support sophisticated filtering mechanisms. This would enable us to construct more complex queries based on previously filtered data, further enhancing the effectiveness and responsiveness of the cache service.

REFERENCES

[1] 2013. How Twitter monitors millions of time series. https://www.oreilly.com/content/how-twitter-monitors-millions-of-time-series/.
[2] 2024. InfluxDB. https://www.influxdata.com/.
[3] 2024. InfluxDB design insights and tradeoffs. https://docs.influxdata.com/influxdb/v1/concepts/insights_tradeoffs/
[4] 2024. InfluxDB key concepts. https://docs.influxdata.com/influxdb/v1/concepts/key_concepts/
[5] 2024. Time series database (TSDB) explained. https://www.influxdata.com/time-series-database/
[6] 2022. What Is a time series? https://www.investopedia.com/terms/t/timeseries.asp.
[7] 2024. What is time series data? https://www.influxdata.com/what-is-time-series-data/
[8] 2024. Compare InfluxDB to SQL databases. https://docs.influxdata.com/influxdb/v1/concepts/crosswalk/
[9] 2024. Influx Query Language (InfluxQL). https://docs.influxdata.com/influxdb/v1/query_language/
[10] https://aws.amazon.com/what-is/materialized-view/
[11] Ding, Jialin, Marcus, Ryan, Kipf, Andreas, Nathan, Vikram, Nrusimha, Aniruddha, Vaidya, Kapil, van Renen, Alexander, & Kraska, Tim. SageDB: An Instance-Optimized Data Analytics System. Proceedings of the VLDB Endowment, 15 (13). Retrieved from https://par.nsf.gov/biblio/10413734. https://doi.org/10.14778/3565838.3565857
[12] Jian Liu, Kefei Wang, and Feng Chen. 2021. TSCache: an efficient flash-based caching scheme for time-series data workloads. Proc. VLDB Endow. 14, 13 (September 2021), 3253–3266. https://doi.org/10.14778/3484224.3484225
[13] https://roman.pt/posts/time-series-caching/
[14] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14). Association for Computing Machinery, New York, NY, USA, Article 16, 1–14. https://doi.org/10.1145/2592798.2592804