A Novel Caching Framework for Efficient Time-Series Analytics

Van Long Nguyen Huu(✉) and An Tran Cong

Can Tho University, Can Tho, Vietnam
{nhvlong,tcan}@ctu.edu.vn

Abstract. Time-series databases (TSDBs) have become essential in various time-series data applications. As the size of time-series datasets continues to grow rapidly, efficient system solutions are needed to manage this data effectively. One promising approach is caching systems that reduce response times and alleviate traffic burdens on back-end TSDBs. However, conventional caching strategies are often suboptimal for handling time-series data. To address this challenge, we propose SEMECAF (SEmantic-aware and MEtadata-driven CAching Framework), a novel caching framework designed to enhance the efficiency of time-series data management. SEMECAF introduces three key innovations: a cache framework, a semantic-aware processing mechanism, and metadata-driven management. Experimental results show that a lightweight implementation of SEMECAF significantly outperforms the baseline approaches across various performance metrics.

Keywords: Semantic-aware · Time-Series Databases · Metadata

1 Introduction

In recent years, time-series databases (TSDBs) have become essential components in various applications [16,17], including data analysis [21,26], Internet of Things (IoTs) monitoring [19,29], and system diagnosis [16,17]. Time-series data in TSDBs consist of *data points* recorded over time [2,3], such as sensor readings or stock prices. This characteristic makes time-series data valuable, allowing us to track behaviors and trends in dynamic information over time. However, managing time-series datasets' enormous and rapidly growing size requires a highly efficient system design. For instance, Uber's modern metrics platform, M3 [1], regularly ingests over 100 million datapoints per minute and serves hundreds of thousands of PromQL queries per second at sub-second latencies. Therefore, implementing high-speed query processing is both critical and challenging. One well-known solution is to utilize a *caching system* to reduce response times and traffic overhead to back-end TSDBs. When data is available in the cache, a cache hit occurs, allowing the cache to respond quickly to queries without communicating with the TSDBs. This approach significantly alleviates the load on TSDBs, improving resource efficiency and reducing processing latency.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15649, pp. 285–302, 2025.
https://doi.org/10.1007/978-3-031-96997-3_18

1.1 Challenges and Opportunities

Due to the unique characteristics of time-series datasets [2,3], traditional caching methods are not optimized for their handling. By examining these properties, we can uncover both the challenges and opportunities for our proposal: a novel approach to the caching system.

Time-Series Data. Time-series data is collected over time intervals to provide insights into past events. This type of data should be considered as *write-once* and *append-only* [2,3]. Therefore, the motivation for cache design becomes straightforward: a *read-only cache* is sufficient.

Query Processing. The predominance of range queries in TSDBs [18] presents several limitations for conventional caches, including the handling of individual data objects or blocks, limited matching cases and overlap between queries. In contrast, a semantic-aware approach [24] can address these issues effectively due to its features: *range query-oriented* processing, management of data groups (referred to as *segments*), and the ability to handle *partial hits*. Consequently, a semantic-aware cache is a strong candidate for our design.

Highly Hot Spots and Skew. Range queries in time-series data applications often focus on specific *hot spots* in the data to analyze behaviors or trends [18]. The percentage of total queries that access these hot spots defines the *skew* of the workload. Most queries follow these patterns as hot spots may arise across the entire dataset. Thus, a semantic-aware design can effectively leverage high hot spots and skew leading to improved space utilization and higher hit ratios for the cache.

1.2 Semantic-Aware and Metadata-Driven Caching Framework

This paper proposes a *SEmantic-aware and MEtadata-driven CAching Framework* called *SEMECAF* that can improve the performance of time-series range queries over TSDBs. In particular, SEMECAF leverages the *semantic awareness* between range queries and manages this property efficiently in the cache with a *metadata-driven* approach. It also provides adaptability with different workload patterns, enabling dynamic optimizations based on observed behaviours of queries. The motivation is primarily to support the range of queries, as they are dominant in time-series data.

SEMECAF can be seen as a *(cache) logic layer* on top of efficient in-memory caching platforms, such as Redis [5] or MemCached [4]. More precisely, SEMECAF handles all the interactions with TSDBs and removes the cache platform from the critical path with TSDBs. This way, the input/output (I/O) performance and management costs are optimized while maintaining a simplified design.

The design can start with a baseline description (e.g., fundamental functionalities and features); then, it could be easily expandable as a *pay-to-go* caching service. Regarding to the modern caching platforms, SEMECAF supports *key-value storage* and provides additional information in forms of *metadata*. Thus,

SEMECAF can be seen as the best of both worlds, where it supports a semantic-aware caching mechanism and meta-data-driven segment management. We summarize the contributions of SEMECAF as follows:

1. *A caching framework.* We provide a read-only caching framework that can be seen as an expandable logic layer on top of caching platforms since time-series data is immutable and append-only. The cache design is simplified by using pay-as-you-go services.
2. *Semantic awareness in processing and caching.* We present segments of ranges of related data points to manage the cache. By matching segments with the interval bounds of incoming queries, our cache can handle partial hits to reduce communication overhead with TSDBs.
3. *Cache management with metadata.* The conventional caches have several management challenges: maintenance of the overhead, filtering latency, storage cost, and query-specific optimization. Our proposal presents a hash-based structure that leverages a composite key and metadata to quickly filter out irrelevant data points in the pool and optimize lookup performance.

Building an efficient cache layer is highly complex and demands addressing many problems; thus, we mainly focus on the core implementations in this paper. Other expansions or optimizations, such as effective replacement policy, low-cost compaction of segments, enhanced partial hit handling, etc., should be researched further. In other words, our proposal can debut as a lightweight framework and create a foundation for conducting further studies.

To evaluate the performance of SEMECAF, we deployed it on top of Redis and used InfluxDB [12] as the back-end TSDB. The experimental results show that our lightweight version of SEMECAF outperforms the baseline approaches (e.g., without cache). SEMECAF can achieve up to 87.1% hit ratio, 35.2 MB/sec for throughput, and accelerate up to 16.9 times the response.

The rest of this paper is organized as follows. Section 2 gives the preliminaries. Section 3 presents the related work. Section 4 describes the design. Section 5 presents our experimental results. The final section concludes this paper.

2 Preliminaries

2.1 TSDB for Data Analytics

Among the TSDBs for data analytics, InfluxDB [12] is explicitly designed for handling large-scale time-series data, including metrics, events, and analytics. InfluxDB stands out from the other options due to its efficient storage engine, rich query language, and ease of integration into diverse ecosystems. Thus, it can be seen as a target for integrating our framework in this paper. To illustrate how InfluxDB works, we present an example as follows: InfluxDB organizes data into *measurements*, which can be considered equivalent to tables in relational databases. Each measurement consists of: *tags*, *fields* and *timestamps*. Tags are the *key-value* pairs indexed for filtering and searching. Fields are key-value

pairs for storing actual data points and not indexed. Meanwhile, timestamps are when the data point was recorded, which is critical for ordering and querying. Moreover, data points in InfluxDB are stored and sorted in *shards* for efficient compression and retrieval. Last, a *series* represents a unique combination of a measurement and its associated tags. More precisely, each series corresponds to a logical stream of data points that share these identifiers. To illustrate how InfluxDB works, we present an example as follows:

```
environment,location=London,sensor=temp value=22.5,humidity=60 1672
environment,location=London,sensor=air value=50        1673
environment,location=Paris,sensor=temp value=19.0,humidity=55  1674
```

According to this example, InfluxDB manages them in the forms of:

- Measurement: *environment*
- Tags: *location* (e.g., London), *sensor* (e.g., temp)
- Fields: *value*, *humidity*.
- Timestamps: represented in nanoseconds (e.g., 1672).
- Shards: all data points belong to the same shard if their timestamps fall within the same duration.
- Series: there are several distinct series. For example, [*environment*, *location* = London, *sensor* = temp].

2.2 In-Memory Caching Platform

Redis is a robust, in-memory data caching platform that operates on generic key-value pairs and supports advanced data structures (i.e., hash). Leveraging these benefits, the logic layer of our design can provide quick access to *segments* that contain data points within a bound of interval time. More precisely, the keys encode the bound of interval time that represents the semantic awareness of relevant queries. For example, a segment can be defined with a key that includes the device ID and the time range of a query, like $device123 : 2024 - 01 - 01 : 2024 - 01 - 15$. Generally, using particular data structures, like *Hash* or *Stored Sets*, Redis allows us to store segments compactly. Thus, it reduces memory consumption, especially when managing many segments. In particular, the hash-based approach contains all the segments related to a specific time series or interval stored as fields under a single key. In this way, the cache significantly reduces the number of keys it needs to manage, thus speeding up operations like scanning and matching segments. Hash fields can be presented through the below example:

```
Key: device123
Hash Fields:
   2024-01-01_to_2024-01-15 -> [Data for this range]
   2024-01-16_to_2024-01-31 -> [Data for this range]
```

Although Redis facilitates the workload of our design, it is not a constraint of the framework we propose. Indeed, the framework's contributions, such as a semantic-aware layer, data organization, and partial hit handling, are still suitable for redistribution to other caching platforms with minimal adjustments.

2.3 Semantic-Aware Processing and Caching

The advantage of semantic-aware caching is twofold. First, segments (e.g., subsets of data) can be much smaller than the original object or block, resulting in better cache space utilization and a higher hit ratio. Second, semantic-aware caching allows for handling partial hits (as shown in Fig. 1), where a part of a query (i.e., sub-query) can be answered quickly. To illustrate how the semantic-aware cache works, we use a typical SQL-like range query as shown below:

```
SELECT temp
FROM sensors
WHERE timestamp
BETWEEN '2024-01-01' AND '2024-01-31';
```

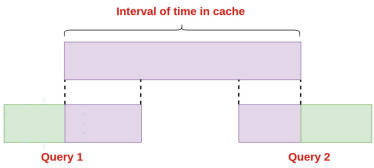


Fig. 1. Partial hits in cache.

The semantic-aware cache has a key: $temp : sensorA : 2024-01-01_to_2024-01-15$. It can detect that the range $2024-01-16$ to $2024-01-31$ is missing due to the interval bound of SQL-like query. Meanwhile, the range from $2024-01-01$ to $2024-01-15$ can be retrieved quickly. Thus, it fetches the remaining data from the TSDBs by generating a new sub-query that covers the missing range $2024-01-16$ to $2024-01-31$. Ultimately, the cache will update a new segment with the new key: $temper : sensorA : 2024-01-16_to_2024-01-31$. Concurrently, the application merges the data from the cache and server to get the final result.

3 Related Work

The basic idea behind of our approach is to accelerate the query processing by leveraging the innovations of TSDB and in-memory caching platform. Then, we build a cache logic layer that sits between them. In this section, we present the related work from three research areas: TSDB contributions, in-memory caching platform and semantic-awareness.

TSDB Contributions. Data analytics in InfluxDB focuses on optimizing data management to improve data ingestion and retrieval performance [2,12]. We have seen other approaches, such as data visualization [9], data sampling and summarizing [18], in-memory processing [16,17]. Since SEMECAF focuses on data analytics, we choose InfluxDB as our back-end database.

Caching Platform. Caching has been widely studied to reduce the latency of databases and TSDBs [27]. We have seen some commercial products implemented on fast local storage (e.g., Solid State Drives SSDs), such as DataBricks [13] and Alluxio [7]. In contrast, Druid [6] uses an external cache to manage the data (i.e., Memcached [4]) in the form of an arbitrary chunking. The main

drawback of such an approach is that it can cause high overhead and inefficient caching. Our framework is based on in-memory caching caching platform Redis since it provides advanced data structures (e.g., hashes, sorted sets, lists) to manage data points with semantic-awareness and metadata-driven.

Semantic-Awareness. Semantic-aware caching has a large body of work [22–24,27]. In recent years, we have witnessed the semantic-aware concept implemented on TSDBs in different works. Wangthammang et al. [28] provide a cache mechanism on top of Memcached to reduce the query time of OpenTSDB. However, their work disregards the issues of data management and replacement policy that can significantly increase performance. Zang et al. [30] propose a novel lightweight semantic cache mechanism for Cortex [8]. One of the most significant disadvantages of such an approach is that the cache is in the critical path of TSDB. Another solution, TSCache [25], proposes a flash-based caching scheme that provides parallel I/O operations on flash SSD. In particular, this work is tailored mainly for InfluxDB, which needs an API to bridge the gap between InfluxDB and the client. Since using the flash SSD, their design raises the question of complexity and scalability. In contrast, SEMECAF leverages Redis' innovations to implement semantic-aware concepts as the logic layer.

4 Prototype Design

This paper presents a SEmantic-aware and MEtadata-driven CAching Framework, called SEMECAF, to accelerate the query range processing. Figure 2 illustrates the workflow between the logic layer (i.e., SEMECAF), storage layer (e.g., InfluxDB), and caching platform (i.e., Redis).

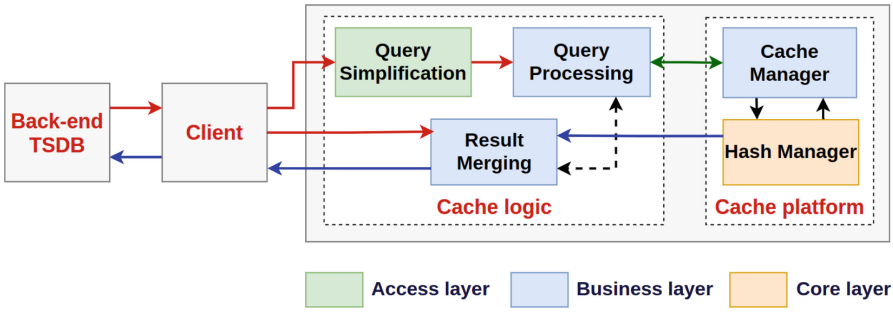


Fig. 2. Architecture of SEMECAF with the major components.

For each query, SEMECAF can be seen as the access point. After a conversation with the cache, if data is found (e.g., *hit*), the logic layer retrieves and returns the data quickly to the user. Otherwise, the logic layer receives a status, e.g., *miss*, and needs to drive the query to the TSDB to get the missing data. After that, the data is forwarded from SEMECAF to the cache and

stored to serve future queries. SEMECAF handles all the principal interactions on both sides, storage and cache, by integrating a logic layer. Therefore, applying SEMECAF in practice is smooth and easy without modifying the TSDB and cache platform.

4.1 Architecture Overview

SEMECAF is described through three main layers: *access layer*, *business layer*, and *core layer*, as illustrated in Fig. 2. In particular, each layer contains one or several components: *Query simplification* in the access layer, *Query processing*, *Cache manager*, and *Result Merging* in the business layer, and *Hash manager* for the core layer. In detail, these components consist of sub-procedures that can be wrapped into the relevant units. For example, *Query processing* can be divided into several units: handling partial hits and generating relevant sub-queries can be grouped into one unit, named *query handling*. Meanwhile, the Cache contains some units, such as *replacement policy* and *basic operations* (e.g., get and retrieve).

We summarize the mission of each component as follows: (1) Query simplification: transforms diverse query formats into a canonical form to reduce redundancy and improve hit rates; (2) Query processing: handles partial hits to generate precise sub-queries and merge the answers into the final result; (3) Cache manager: provides primitive functions to manipulate data (i.e., set, get or find), allows duplicate or non-duplicate segments, and implements a replacement policy; (4) Hash structure manager: exploits the hash-based structure of modern caching platforms with composite keys and metadata-driven segments; and (5) Result merging: bridges the gap between InfluxDB and Redis in the representation of data. These components can interact with each other to accomplish the demands of queries.

4.2 Query Simplification

Time-series queries can contain redundant or overlapping time intervals and unordered dimensions. Simplification ensures consistent query representation, enabling efficient hash-based lookup and reducing computational redundancy.

The Query simplification transforms complex queries into a canonical form by merging overlapping intervals and standardizing the ordering of dimensions. This process minimizes lookup time and prevents unnecessary cache misses caused by syntactic differences in semantically equivalent queries. In other words, it ensures that all subsequent operations (e.g., partial hit handling in Query processing) are optimized. We present the Query simplification in Algorithm 1.

To clarify the workflow of Algorithm 1, lets take an example. Assuming that an incoming query Q contains $D = \{sensor, location\}$ and $T = \{[10, 15], [12, 20], [25, 30]\}$. In Step 1, since T is already ordered, $T_s = T$. Then, we merge the overlapping intervals in T_s, the result is: $T' = \{[10, 20], [25, 30]\}$. While in Step 2, we get the $D' = \{location, sensor\}$. We have the result as $Q' = (\{[10, 20], [25, 30]\}, \{location, sensor\})$ at the end of the algorithm.

Algorithm 1. Query Simplification

Input: Query $Q = (D, T)$, where $D = \{d_1, d_2, \ldots, d_k\}$ is the set of dimensions and $T = \{[t_{1,1}, t_{1,2}], [t_{2,1}, t_{2,2}], \ldots, [t_{n,1}, t_{n,2}]\}$ is the set of time intervals
Output: Simplified query $Q' = (D', T')$

Sort T by $t_{i,1}$, obtaining $T_s = \{[t_{i,1}, t_{i,2}] \mid i \in [1, n]\}$.
Initialize $T' \leftarrow \emptyset$.
for $i = 1$ to n **do**
    if $T' = \emptyset$ **or** $t_{i,1} > \text{end}(T'_{\text{last}})$ **then**
        Append $T_{s,i}$ to T'.
    else
        Merge $T_{s,i}$ with T'_{last}: $T'_{\text{last}} \leftarrow [t_{\text{start}}(T'_{\text{last}}), \max(t_{\text{end}}(T'_{\text{last}}), t_{i,2})]$
    end if
end for
Sort dimensions D lexicographically: $D' \leftarrow \text{sort}(D, \text{key} = d_i)$
$Q' \leftarrow (D', T')$.
return Q'.

4.3 Query Processing

Generally, Query processing takes responsibility for: (1) handling hits, misses, and partial hits; (2) generating sub-queries depending on the overlap with cached segments; (3) constructing and maintaining the execution of sub-queries for the uncovered data ranges; (4) combining the results to form the final answer with Result Merging. Note that Query processing cannot work independently. Indeed, the cache manager provides a set of operations to manipulate the data segments on the caching platform, such as *set* (store), *get* (retrieve), *find* (match), and others. For example, *IdentifyCachedSegments* of Query processing uses the *find* function of the cache manager to match the overlapping data segments that are represented in the form of *a hash*.

Partial Hit Handling. Optimized partial hit manipulation allows for the exploitation of diverse query patterns effectively, even when the data distribution is non-uniform. In other words, this is the core of Query processing that significantly impacts cache performance metrics, such as response time, hit ratio, space overhead, and throughput. Algorithm 2 illustrates how the cache handles the partial hit as the most complex scenario of query processing. Since hits and misses are the baselines of the cache, their discussion can be neglected.

Since this paper mainly focuses on the proofs of concept before going further to tackle the optimizations of the framework, partial hit handling does not take into account these issues: a large set of disjoint time ranges, multiple fragmented sub-queries, granularity of data, and the expensive burden of programming. In other words, we have an *"optimist"* partial hit handling in SEMECAF. Nevertheless, the experiments show that our approach is still better than the baseline solutions.

Algorithm 2. Query processing with partial hits

Input: Query $Q = (d_1, d_2, \ldots, d_k)$, Cache S
Output: Result R

procedure HANDLEQUERY(Q, S)
  $S_c \leftarrow IdentifyCachedSegments(Q, S)$ ▷ Find overlapping cached segments
  $S_{nc} \leftarrow Q - \bigcup S_c$ ▷ Calculate uncovered ranges
  if $S_{nc} \neq \emptyset$ **then**
    $Q_{sub} \leftarrow GenerateSubQueries(S_{nc})$ ▷ Sub-queries for uncovered ranges
    $R_{sub} \leftarrow ExecuteSubQueries(Q_{sub})$ ▷ Execute sub-queries on the backend
  else
    $R_{sub} \leftarrow \emptyset$
  end if
  $R \leftarrow CombineResults(S_c, R_{sub})$ ▷ Merge cached and backend results
  return R
end procedure

Complexity Analysis. The complexity of HandleQuery differs significantly from the old-fashioned storage of segments (i.e., without leveraging hash, composite keys, and metadata). HandleQuery's hash keys point directly to relevant segments, reducing unnecessary comparisons. In contrast, in old-fashioned storage, we must perform a sequential scan through all cached segments to identify the overlaps. Indeed, the complexity of the old-fashioned method is $O(k \cdot n)$, where k is the number of dimensions and n is the number of segments for each dimension. Such complexity creates a significant bottleneck in processing, especially for large datasets. Meanwhile, the approximate complexity of HandleQuery is $O(k+m)$, with m as the number of relevant cached keys. Thus, it can perform significantly better for larger caches and complex queries, such as lookup and overlap identification.

The full expression complexity of Algorithm 2 is: $O(k \log |S| + k \log k + m \cdot |F| + m \cdot q$. We break down the complexity into sub-procedures of HandleQuery:

- $k \log |S|$: cost for identifying cached segments, with $|S|$ being the size of the cache. It is derived from `Intersect`(S_i, Q) in `IdentifyCachedSegments`(Q, S). The Intersect can be calculated by using max and min of interval bounds of time ranges (as can be seen in Algorithm 3).
- $k \log k$: cost of sorting dimensions to ensure canonical order for simplification.
- $m \cdot |F|$: cost of overlap calculation where $|F|$ is the number of fields (i.e., metadata complexity).
- $m \cdot q$: cost of generating sub-queries based on uncovered segments (q is the cost of formulating a single sub-query). It is derived from computing uncovered ranges $\mathcal{U} = Q \setminus \bigcup_{S_i \in \mathcal{S}_c} S_i$ and `FormulateQuery`(U_j, F) of `ExecuteSubQueries`(Q_{sub})) (as can be found in Algorithm 4).

The cost is primarily driven by $\log |S|$, where S is often organized as a hash table or composite key structure for efficient lookup of cached segments. Indeed,

Algorithm 3. Matching between query and cached segments

Require: Query range $Q = [t_{\text{start}}, t_{\text{end}}]$, Cached segments $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$
Ensure: Set of overlapping cached segments $\mathcal{S}_c$
1: Initialize $\mathcal{S}_c \leftarrow \emptyset$
2: **for all** $S_i = [t^i_{\text{start}}, t^i_{\text{end}}]$ in $\mathcal{S}$ **do**
3: Compute `Intersect`$(S_i, Q) = [\max(t_{\text{start}}, t^i_{\text{start}}), \min(t_{\text{end}}, t^i_{\text{end}})]$
4: **if** $\max(t_{\text{start}}, t^i_{\text{start}}) \leq \min(t_{\text{end}}, t^i_{\text{end}})$ **then**
5: Add S_i to $\mathcal{S}_c$
6: **end if**
7: **end for**
8: **return** $\mathcal{S}_c$

Algorithm 4. Generate Sub-Queries with formulation

Require: Query range $Q = [t_{\text{start}}, t_{\text{end}}]$, Overlapping cached segments $\mathcal{S}_c$, Original query F (e.g., dimensions, filters).
Ensure: List of sub-queries $\mathcal{Q}_{\text{sub}}$
1: Initialize $\mathcal{Q}_{\text{sub}} \leftarrow \emptyset$
2: Compute uncovered ranges $\mathcal{U} = Q \setminus \bigcup_{S_i \in \mathcal{S}_c} S_i$
3: **for all** $U_j = [t^j_{\text{start}}, t^j_{\text{end}}]$ in $\mathcal{U}$ **do**
4: query $\leftarrow$ `FormulateQuery`(U_j, F)
5: Add query to $\mathcal{Q}_{\text{sub}}$
6: **end for**
7: **return** $\mathcal{Q}_{\text{sub}}$

$\log |S|$ lookups significantly outperform linear scans $O(n)$. Additionally, more considerable m leads to more merging and sub-query generation. As a result, handling overlaps is expensive. Lastly, sub-query generation q involves splitting uncovered ranges into one or more backend queries, requiring computational effort proportional to the number of such ranges. In conclusion, the performance of our cache outperforms the baseline and old-fashioned semantic approaches.

4.4 Cache Manager

A side-by-side component of query processing in our framework is the cache manager. The Cache manager provides a library of primitive functions, as shown in Table 1. Interestingly, they are designed with the same input: composite key k. Such a key is created and managed by a hash-based structure within the hash manager. The composite keys allow efficient lookup of relevant data segments and minimize the space overhead of the cache. In our design, the cache manager works on top of the hash manager since the primitive functions can leverage the tailored structures that effectively represent the cached data. By this way, Cache manager can focus on the cache operations and mechanism, meanwhile data structure from basic to complex are encapsulated in Hash manager.

Note that this table does not present the complete list of primitive functions. For instance, we also implement the primitives for creating composite keys (i.e.,

Table 1. Some of the primitive functions in Cache manager

Function	Purpose	Input	Output
`CacheGet`	Retrieve a value by its composite key	Composite key k	Cached segment v or `null`
`CacheSet`	Store a value with a composite key	Composite key k, Segment v	Confirmation of insertion
`CacheFind`	Find all overlapping segments for a query range	Query range Q	Set of overlapping segments S_c
`CacheMatch`	Check if a specific key exists in the cache	Composite key k	Boolean: `true` or `false`
`CacheUpdate`	Update a cached segment or its metadata	Composite key k, New segment v	Confirmation of update
`CacheDelete`	Remove a segment by its composite key	Composite key k	Confirmation of deletion

KeyCreate), merging adjacent segments (i.e., CacheCoalesce), etc. All of these functions can be easily implemented using a variety of Redis client frameworks, such as Jedis [11] or redis-py [14]. Since we are familiar with Java, we chose Jedis, which is designed for performance and ease of use.

Another role of the Cache manager is to deploy a replacement policy and coalescing strategy. This paper applies the Least Recently Used (LRU) replacement policy and a non-coalescing strategy. Although they are not the most optimized options for our cache, they are reasonable enough to evaluate our proposal. Some further optimizations could be implemented in the cache manager, such as using a threshold to reduce the complexity of cache lookup, estimating the size of a segment as metadata and others. All of these contributions aim to present an automated and adaptable cache manager.

4.5 Hash Manager

The Hash Manager provides the customized hash with composite keys and metadata. Such an approach is a significant enhancement over the native hash of Redis, tailored to support (multidimensional) range queries efficiently.

Unlike simple keys, a composite key encodes multiple dimensions (e.g., *time*, *location*, *sensor_type* regarding *Temperature data in a city*) and ranges of a specific data segment into a unique identifier. In short, a composite key ensures *data isolation* and *semantic context* of the cached segment. In this way, composite keys allow efficient lookup of relevant data segments.

We also wrap the other descriptions into metadata in our hash-based structure, such as *start_time*, *end_time* and *average_density* based on the type of cached data. The most significant advantage is that it allows more straightforward and faster range lookup through efficient pruning of irrelevant segments during Query processing, especially in the case of multidimensional data. Moreover, metadata is essential for enabling the intelligent decision-making of Cache managers, such as using thresholds, estimating size, and computing overlapping segments. More precisely, it allows for handling overlapping and uncovered ranges where we want to coalesce the segments.

To emphasize the benefits of our hash-based structure, we present side by side the two approaches below.

```
--- NATIVE HASH ---
Hash Key: temperature
Hash Value:
 field: "London:T1"
 value: "{data: [12.5, 13.0, 14.0]}"
 field: "London:T2"
 value: "{data: [15.0, 15.5]}"
```

```
--- CUSTOM HASH ---
Hash Key: London:temperature
Hash Value:
 field: "London:T1-T2"
 value: "{stime: T1, etime: T2,
  dens: 10, data: [12.5, 13.0]}"
 field: "London:T2-T3"
 value: "{stime: T2, etime: T3,
  dens: 10, data: [14.0, 15.0]}"
```

In the native hash, all temperature data might be grouped under the key *"temperature"*. Each field within the hash represents a unique identifier (e.g., *"London:T1"*), where $T1$ is the start of the first time interval. Similarly, the second field contains data for the second time interval. In fact, it does not contain any metadata. This approach generates a large set of segments to be checked in query processing. It is inefficient because we require examining all fields to identify overlaps with the query. Moreover, time intervals and their relationships (e.g., overlap, adjacency) are not explicitly encoded.

In contrast, the custom structure creates a hierarchical and context-rich composite key of multiple dimensions (i.e., *"London:temperature"*). Such a key is used to narrow down the relevant fields efficiently. Moreover, in the growth of multidimensional, number of devices (e.g., $sensor1$, $sensor2$) and measurements (e.g., *pressure*), this approach significantly reduces the complexity of Query processing. We also notice that each field in the hash is tied to a specific time interval (e.g., *"London:T1-T2"*) and includes metadata for query optimization (e.g., *"stime"*, *"dens"*). Thus, the handling of overlapping or adjacent segments could be simplified.

Note that although we do not have any indexing to optimize the lookups in the cache, the custom hash provides some of the same benefits. More precisely, by organizing data under a structured key and metadata schema, retrieval is faster and more targeted. Moreover, using metadata, it can mimic the index statistics used in query execution plans for partial hits and sub-query handling. In essence, the custom hash works as a lightweight lookup mechanism akin to an index.

4.6 Result Merging

Redis typically serves data in key-value pairs, which is not directly aligned with the native format of query results from InfluxDB. In contrast, InfluxDB delivers query results in a structured format tailored for time series data, including fields such as timestamps, tags, and field values. Thus, we must normalize and bridge the gap between the two platforms using a Translator Application Programming Interface (API) which has the complexity $O((n_c+n_b)$. n_c is the number of cached segments retrieved and n_b is the number of TSDB rows. Since the final result should be sorted and merged from intermediate results of back-end server and cache, we define a component called Result Merging which contains Translator API. Thus, the total complexity of such module is: $O((n_c + n_b)\log(n_c + n_b))$.

Regarding to this complexity, the current Result Merging of SEMECAF scales well for moderate-sized queries, but issues may arise when the number of cached and TSDB results grows significantly.

5 Experiments

5.1 Experimental Setups

Environments. In particular, InfluxDB 1.11.8 is deployed on a server with a 64-bit Ubuntu 22.04 LTS system and configured with an 8-core Intel Xeon Gold 5118 CPU running at 2.30 GHz and 64 GB RAM. Redis Community Edition is installed on another PC with the same system and configuration. The client to drive the workloads is a 64-bit Ubuntu 22.04 LTS system and a 4-core Intel Core^{TM} i5-8265U CPU running at 1.60GHz and 16 GB RAM. Note that we first create a database and load data points into the database, and in each run, we start with the same loaded database.

Datasets. Two real-world time-series datasets are used: Traffic [15] and Sensor [10]. Since they are small, we need to enlarge them by repeating their data multiple times for a longer duration while keeping the other metrics (e.g., sampling rate, fields, etc.). We then have about 6 GB (82, 419, 304 data points) of Traffic with nine attributes and 5 GB (63, 775, 979 data points) of Sensor with eight attributes.

Workloads. We use time-series benchmark YCSB-TS [20] to generate queries with different time ranges. The length of time range (i.e., $[time_start, time_end]$) follows the Zipfian distribution between *maxscanlength* and *minscanlength* as defined in the configuration. Our workload consists of 1,000 queries where each of them is varied from one minute to one week. The average size of the query is 20.15 KB and 15.31 KB for Traffic and Sensor, respectively.

To evaluate the performance in detail, we also present three workloads on Traffic: Small, Medium, and Large, where the time range of queries is 1 min to 1 h, 1 h to 1 day, and 1 day to 1 week, respectively. The average size of the query is 5.78 KB, 66.19 KB, and 357.53 KB. The cache size is set to 10% of the size of the workload's data set.

Metrics and Scenarios. We measure the performance using three key metrics: *hit ratio*, *throughput*, and *response time*. We compare SEMECAF with three other system solutions: (1) *W-Cache*: without using any cache; (2) *B-Cache*: using a fixed-size time unit (i.e., block) indexed by its query and time length in Redis; and (3) *OS-Cache*: old-fashioned semantic cache on Redis without using composite keys and metadata.

5.2 Comparison

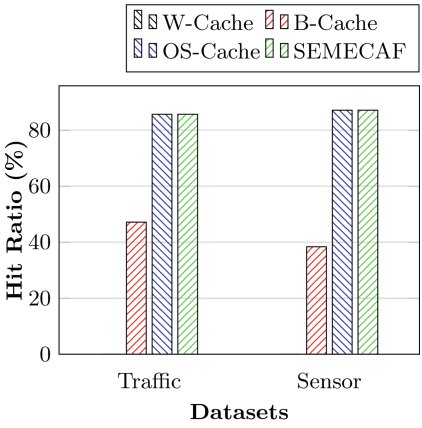
With Traffic and Sensor. Figure 3 shows the performance of four solutions in various metrics: hit ratio, throughput, and response time. SEMECAF achieves the best overall performance across the two workloads. The hit ratio of SEMECAF can achieve up to $85.7 - 87.1\%$, which is better than B-Cache (i.e.,

38.4 − 47.2%). Note that the W-Cache has no hit ratio since no cache is deployed. Meanwhile, SEMECAF and OS-Cache share the same hit ratio thanks to their semantic-aware approach.

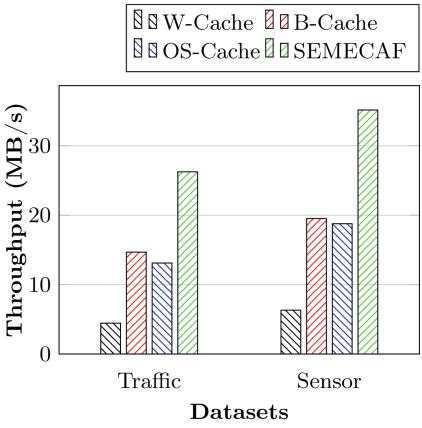
Regarding the throughput, SEMECAF shows the best results compared to other solutions. More precisely, it increases the throughput by a factor of $7.6 - 10.0$ when compared with W-Cache. Moreover, we note that this throughput also leads to an increase by a factor of up to 2.0 when compared with B-Cache and OS-Cache. We notice that by using metadata-driven techniques with a customized hash and composite key, SEMECAF can overcome the bottleneck of OS-Cache (i.e., a large set of segments to be checked) by quickly filtering irrelevant segments in the lookup procedure. Indeed, because of this bottleneck, OS-Cache has been surpassed by B-Cache. Moreover, when the query is more complex (e.g., multidimensional), OS-Cache could fall behind in the race against B-Cache. Thus, SEMECAF is the best solution for replacing OS-Cache.

The key metric of the experiment to prove the benefits of SEMECAF is the response time. In particular, SEMECAF shows a speed-up by a factor of $2.1 - 2.6$ and $2.5 - 2.7$ when compared with B-Cache and OS-Cache, respectively. Moreover, the response time of SEMECAF is far better than that of the W-Cache, where we see the speed-up can achieve up to $15.0 - 16.9$ times. Note that the speed-up can vary if we choose a different workload size. Thus, to understand the performance gains in more detail, we compare SEMECAF and other solutions under the Traffic data set in Fig. 4.

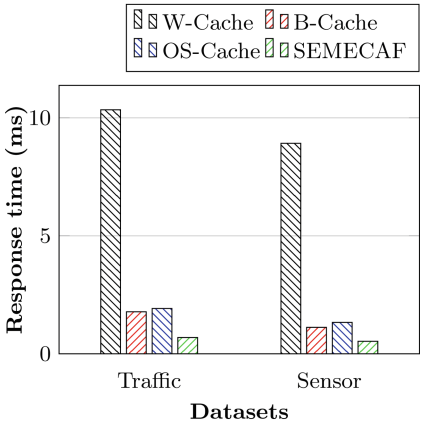
With Various Size of Traffic. In this experiment, the block size of B-Cache is fixed at a 1-hour unit. Note that the more fine-grained the chunking is, the higher the hit ratio, but also the higher the overhead can be. For different workload sizes, the semantic-aware approach's hit ratio is solid. More precisely, OS-Cache and SEMECAF always provide a hit ratio of over 80%. Meanwhile, in B-Cache with a small workload, the requests cannot benefit from the cache due to the fixed-size block (i.e., only 46.2%). When the size of the block increases, the hit ratio can significantly decrease or even drop to zero. B-Cache achieves more comparable hit ratios in the case of medium and high workloads, with 60.1% and 90.2%, respectively. Indeed, its hit ratio is higher than SEMECAF's in large workloads. The reason is that B-Cache can gain more partial hits with heavier workloads. However, due to the complexity of partial hit handling, which incurs severe I/O amplification, the trade-off appears in throughput and response time. SEMECAF is $1.2 - 1.8$ times higher than B-Cache in throughput, and its response time is $2.4 - 3.1$ times faster, thanks to its pseudo-hash-based indexing and irrelevant segment filtering. SEMECAF is also superior to W-Cache by $9.9 - 16.8$ times. It is worth noting that the number of requests is significantly amplified in the case of multidimensional workloads, resulting in an overload on TSDB. In conclusion, this result shows that using SEMECAF is much more efficient.



(a) Hit ratios (%).

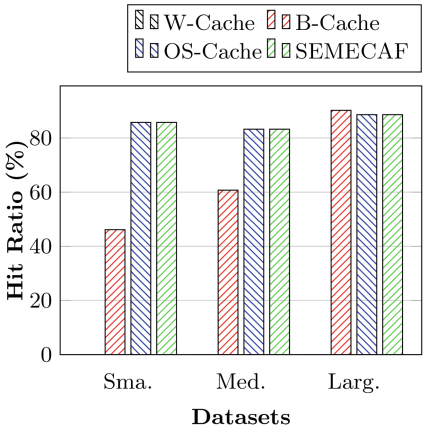


(b) Throughputs (MB/sec).

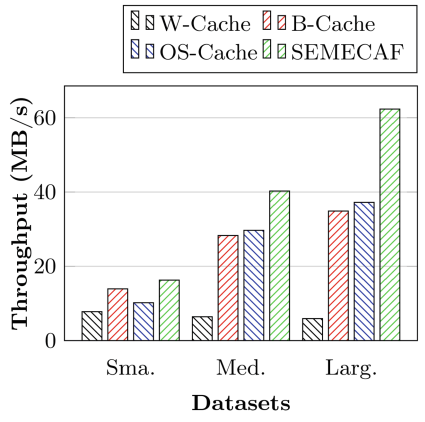


(c) Response times (ms).

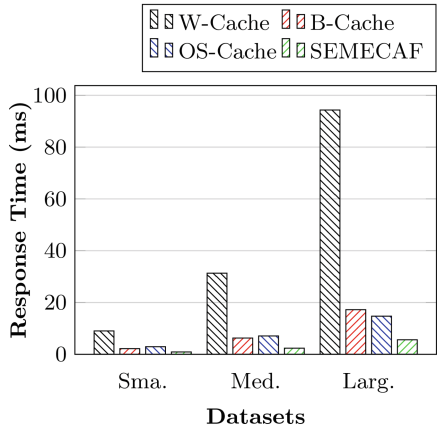
Fig. 3. Evaluations with two datasets: Traffic and Sensor.



(a) Hit ratios (%).



(b) Throughputs (MB/sec).



(c) Response times (ms).

Fig. 4. Evaluations with various size of workloads on Traffic dataset.

6 Conclusion and Future Works

In this paper, we present SEMECAF, an efficient in-memory semantic-aware and metadata-driven caching solution that accelerates the range of queries in TSDBs. The results show that SEMECAF can significantly improve the effectiveness and efficiency of query processing. Since this paper mainly focuses on the proofs of concepts, we could address the following issues in our future works, such as multiple indexing scheme, optimized replacement, coalescing with threshold and size estimation, granularity as metadata, multiple disjoint time ranges in partial hit, overhead of CPU performance, and multiple clients performance.

Acknowledgment. The authors would like to thank colleagues of College of Information and Communication Technology, Can Tho University for their insightful comments.

References

1. M3: Metrics, monitoring, and metadata at scale. https://github.com/m3db/m3 (2018)
2. Time series data analysis? https://www.influxdata.com/what-is-time-series-data/ (2021)
3. What is a time series and how is it used to analyze data? https://www.investopedia.com/terms/t/timeseries.asp (2021)
4. What is memcached? https://memcached.org/ (2021)
5. Your app is about to get faster. https://redis.io/ (2021)
6. Apache druid. https://druid.apache.org/ (2023)
7. Alluxio caching. https://docs.alluxio.io/ee-da/user/stable/en/core-services/Caching.html (2024)
8. Cortex documentation. https://cortexmetrics.io/docs/ (2024)
9. Graphite. https://graphiteapp.org/ (2024)
10. Intel lab data. https://db.csail.mit.edu/labdata/labdata.html (2024)
11. Java client for redis. https://github.com/redis/jedis (2024)
12. The leading platform for time series data. https://www.influxdata.com/ (2024)
13. Optimize performance with caching on databricks. https://docs.databricks.com/en/optimizations/disk-cache.html (2024)
14. Python client for redis. https://pypi.org/project/redis/ (2024)
15. Traffic dataset collection. http://iot.ee.surrey.ac.uk:8080/datasets.html (2024)
16. Abraham, L., Allen, J., Barykin, O., Borkar, V., Chopra, B., Gerea, C., Merl, D., Metzler, J., Reiss, D., Subramanian, S., Wiener, J.L., Zed, O.: Scuba: diving into data at Facebook. Proc. VLDB Endow. **6**(11), 1057–1067 (2013)
17. Adams, C., et al.: Monarch: Google's planet-scale in-memory time series database. Proc. VLDB Endow. **13**(12), 3181–3194 (2020)
18. Agrawal, N., Vulimiri, A.: Low-latency analytics on colossal data streams with summarystore. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 647–664. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017)

19. Aljawarneh, S., Radhakrishna, V., Kumar, P.V., Janaki, V.: A similarity measure for temporal pattern discovery in time series data generated by IoT. In: 2016 International Conference on Engineering & MIS (ICEMIS), pp. 1–4 (2016)
20. with support for time series data bases, Y.T.: https://tsdbbench.github.io/YCSB-TS/ (2024)
21. Cui, H., Keeton, K., Roy, I., Viswanathan, K., Ganger, G.R.: Using data transformations for low-latency time series analysis. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 395–407. SoCC '15, Association for Computing Machinery, New York, NY, USA (2015)
22. Godfrey, P., Gryz, J.: Answering queries by semantic caches. In: Proceedings of the 10th International Conference on Database and Expert Systems Applications, pp. 485–498. DEXA '99, Springer-Verlag, Berlin, Heidelberg (1999)
23. Huu, V., Lallet, J., Casseau, E., d'Orazio, L.: MASCARA (modular semantic caching framework) towards FPGA acceleration for IoT security monitoring. Open J. Internet Things **6**(1), 14–23 (2020)
24. Jónsson, B.T., Arinbjarnar, M., Þórsson, B., Franklin, M.J., Srivastava, D.: Performance and overhead of semantic cache management. ACM Trans. Internet Technol. **6**(3), 302–331 (2006). https://doi.org/10.1145/1151087.1151091
25. Liu, J., Wang, K., Chen, F.: TSCache: an efficient flash-based caching scheme for time-series data workloads. Proc. VLDB Endow. **14**(13), 3253–3266 (2021)
26. Rabkin, A., Arye, M., Sen, S., Pai, V.S., Freedman, M.J.: Aggregation and degradation in JetStream: streaming analytics in the wide area. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, pp. 275–288. NSDI'14, USENIX Association, USA (2014)
27. Stonebraker, M., Jhingran, A., Goh, J., Potamianos, S.: On rules, procedure, caching and views in data base systems. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 281–290. SIGMOD '90, Association for Computing Machinery, New York, NY, USA (1990)
28. Wangthammang, T., Tandayya, P.: A software cache mechanism for reducing the OpenTSDB query time. In: 2018 18th International Symposium on Communications and Information Technologies (ISCIT), pp. 60–65 (2018)
29. Xu, X., Huang, S., Chen, Y., Browny, K., Halilovicy, I., Lu, W.: TSaaaS: time series analytics as a service on IoT. In: 2014 IEEE International Conference on Web Services, pp. 249–256 (2014)
30. Zhang, K., Wang, Z., Shao, Z.: BSCache: a brisk semantic caching scheme for cloud-based performance monitoring timeseries systems. In: Proceedings of the 51st International Conference on Parallel Processing. ICPP '22, Association for Computing Machinery, New York, NY, USA (2023)