STsCache: An Efficient Semantic Caching Scheme for Time-series Data Workloads Based on Hybrid Storage

Tao Kong
Xidian University
jasonk@stu.xidian.edu.cn

Hui Li*
Xidian University
Yunxi Technology Co., Ltd
hli@xidian.edu.cn

Yuxuan Zhao
Xidian University
yuxzhao@stu.xidian.edu.cn

Liping Li
Xidian University
lipingli@stu.xidian.edu.cn

Xiyue Gao
Xidian University
xygao@xidian.edu.cn

Qilong Wu
Xidian University
qilongwu@stu.xidian.edu.cn

Jiangtao Cui
Xidian University
cuijt@xidian.edu.cn

ABSTRACT

Due to the increasing demand for extreme-scale time-series data workloads in data centers, it is required to build a high-performance semantic caching system that leverages the semantics and results of historical queries to answer time-series queries. Existing caching solutions either ignore the semantics of queries, offering suboptimal performance, or focus only on specific scenarios, providing small-capacity, limited functionality.

In this paper, we summarize the query patterns of time-series data workload and propose the definition of semantic time-series caching for the first time. Accordingly, we present a semantic time-series caching system, STsCache, based on a hybrid storage model with memory and NVMe SSD. We propose a series of optimized strategies, such as slab-based semantic data management, semantic index, semantic value-driven batch eviction, time-aware deduplication insertion, and lazy compaction. We implemented and evaluated STsCache via benchmarks and production environments. STsCache can increase throughput of popular time-series databases (InfluxDB, TimescaleDB) by 4.8-10.8× and reduce latency by 79.9%-93.5%. Compared with the latest time-series caching schemes (TSCache, BSCache), STsCache can increase throughput by 1.5-4.5×, reduce latency by 59.4%-81.9%, and increase hit ratios by 22.5%-82.4%.

PVLDB Reference Format:
Tao Kong, Hui Li, Yuxuan Zhao, Liping Li, Xiyue Gao, Qilong Wu, and Jiangtao Cui. STsCache: An Efficient Semantic Caching Scheme for Time-series Data Workloads Based on Hybrid Storage. PVLDB, 18(9): 2964 - 2977, 2025.
doi:10.14778/3746405.3746421

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ts-lab1024/ts-semantic-caching.

1 INTRODUCTION

Time-series data is a collection of data points arranged in chronological order, commonly used to represent values that change over time [41]. The past decades have witnessed an explosive growth of time-series data in numerous applications, *e.g.,* Internet of Things (*abbr.,* IoT) [30, 68, 78], meteorology [1, 52, 74], financial engineering [27, 64], manufacturing [40, 58, 63], Internet monitoring [21, 46, 67], *etc.* Therefore, time-series databases (*abbr.,* TSDBs) have become essential in data centers for managing massive, rapidly growing time-series data. For most cases, fast response to queries over massive time-series data is fundamental [80]. As time-series queries are highly skewed towards recent data, they tend to overlap more than other queries [22, 53]. Moreover, these queries typically involve time-ranges, tags, and aggregations. In order to reuse query results flexibly, it requires to explore the correlation between the semantics of queries. Thus, designing a large-capacity cache to answer new queries using the semantics and results of past queries is a potent solution to enhance query performance.

Memcached [34], as a representative key-value cache system, proposes to cache frequently accessed data, *i.e., hot data*, in memory, effectively improving query performance. However, key-value cache systems [2, 4, 9, 34, 49, 51] do not work on TSDBs as they cannot accommodate the varying time-ranges in queries. Due to that, TSCache [53], a flash-based caching scheme, adopts a time-range based query interface to match varying time-ranges. It hashes the query content excluding the time-range into a *key* and encodes the data points as the *value*. Obviously, this design loses the relationship between metadata (*e.g., tags*) and the data, which brings two disadvantages. ❶ It cannot provide a flexible *tag-based interface* like popular TSDBs [10, 12], which is used for retrieving and grouping specified time-series data by tags. ❷ It may suffer from a low hit ratio. The cache can only be hit if the hashed query key perfectly matches one in the cache. The flexible parameter combinations in time-series data workloads, especially with tags, result in a large number of unique keys, significantly reducing the hit ratio. To perceive the relationship between metadata and data, Zhang *et al.* [77] proposed to rely on semantic caching. However, as stated by them, it is difficult to achieve that goal without embedded into the target system. Accordingly, they present a caching scheme, BSCache, acting as a built-in component for a performance monitoring system. Notably, it is a pure in-memory caching component designed for a specific system, thus lacking generality and unsuitable for large-capacity caching scenarios. Furthermore, BSCache overlooks the semantic relationships between queries, which hinders its ability to effectively reuse cached data, such as aggregating data in the cache

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746421

to respond to new queries. Given the progress of these efforts, it suggests that an efficient and large-capacity semantic caching scheme designed for time-series workloads, referred to as *semantic time-series caching*, can help alleviate pressure on back-end databases and reduce query time accordingly. However, proposing such a scheme is challenging for the following reasons.

- **Costly semantic matching.** Although semantic matching usually enables better utilization of cache data than key-value matching, it is more complex and incurs higher overhead. In large-capacity caching, the overhead of semantic matching can significantly degrade the performance of the caching system.
- **Memory overhead/Performance trade-offs.** Handling large time-series data volume requires larger capacity caches, which can incur significant memory overhead if entirely built in memory. TSCache proposes to build caches on flash storage, but this can limit the performance because of flash I/O. Current semantic caching efforts haven't balanced memory overhead and overall efficiency.
- **Cache penetration.** Cache penetration refers to queries for non-existent data, resulting in direct access to the back-end database at all times. This degrades system performance and increases the burden on the database.

In this paper, to overcome the aforementioned challenges, we study and summarize the characteristics and query patterns of time-series data, and we propose a formal definition of semantic time-series caching for the first time. In particular, we define *semantic meta* and *semantic series*. The former encapsulates the semantics of time-series queries, while the latter refers to the semantics of the data sources in queries. Based on this formal definition, we propose a semantic time-series caching scheme called STsCache, which is built on a hybrid storage of memory and NVMe SSD. Unlike BSCache, which is binding to a specific system and scenario, STsCache can provide caching services for general TSDBs.

We designed a series of mechanisms to ensure the STsCache performance. ❶ It uses slabs (*i.e.,* contiguous blocks in memory or SSD) to efficiently store data points from the same semantic series in chronological order, leveraging hardware sequential I/O and reducing management costs. ❷ To reduce semantic matching costs, we designed a semantic index using a graph and skip lists, avoiding expensive matching processes and storing lightweightly empty query semantics to prevent cache penetration. ❸ To retain hot data in the cache and keep hotter data in memory, we have quantified the hotness of the data as *semantic values* and designed a batch eviction scheme accordingly to improve cache performance. ❹ To improve cache space utilization, we proposed deduplication insertion algorithms and slab compaction under certain conditions to reduce space fragmentation.

To evaluate performance, we implemented STsCache based on Fatcache [9] and its client based on gomemcache [13], modifying the Golang clients of InfluxDB and TimescaleDB to access both cache and database services. Experimental results on popular TSDB benchmarks show that, compared to advanced TSDBs, STsCache can improve throughput by 4.8-10.8× and reduce latency by 79.9%-93.5%. Compared with the latest time-series cache [53, 77], STsCache can increase throughput by 1.5-4.5×, reduce latency by 59.4%-81.9%, and increase hit ratios by 22.5%-82.4%.

In summary, the major contributions in this work are as follows:

- We propose the formal definition of semantic caching designed specifically for time-series data workloads for the first time. This serves as a foundation for building semantic time-series caches for general TSDBs.
- We propose building a large capacity cache based on the hybrid storage of memory and NVMe SSD, and design a batch eviction scheme based on semantic values. By keeping hotter data in memory, it balances the query performance and the memory overhead of the caching system.
- We propose a novel semantic index. It can effectively store the semantics of queries and the semantic relations between queries. The semantic index can avoid many expensive semantic matching processes. In addition, it can prevent cache penetration by storing the semantics of empty queries.
- We propose a low-cost compaction strategy, *i.e.,* deduplicating insertions and lazy compaction, to improve the utilization rate of the cache space.

The rest of this paper is organized as follows. In Section 2, we introduce the preliminary concepts of TSDBs and review related works. In Section 3, we propose a formal definition of semantic time-series caching. In Section 4, we provide a detailed description of the overall design as well as details for each module of STsCache. The experimental settings and results are reported in Section 5. Finally, Section 6 concludes this paper.

2 PRELIMINARIES AND RELATED WORK

2.1 Preliminaries of TSDBs

TSDBs are specifically designed to efficiently store, manage, and query large volumes of time-series data [47]. Some of the most well-known and widely recognized TSDBs include InfluxDB [10], TimescaleDB [12], Apache IoTDB [7], OpenTSDB [3], Prometheus [6], QuestDB [5], and others [16, 18].

For ease of discussion, we introduce the relevant concepts of TSDBs using the IoT use case in InfluxDB and TSBS (as shown in Figure 1). Each record in the table is called a *point*, which consists of a *measurement*, a *tag set*, a *field set* and a *timestamp*. A *measurement* is similar to a table in relational databases. The *tag set* comprises key-value pairs (*i.e., tag name* and *tag value*) that describe static characteristics of the data source, such as the truck's **name** and **driver**. The field set contains a key-value pair (*i.e.,* field key and field value) that capture values dynamically changing over time, such as **velocity** and **heading**.

TSDBs employ the notion *series* to refer to a group of points that share the same *measurement*, *tag set* and *field key*. For example, the first two datapoints in Figure 1 belong to the same series, *i.e.,* **series1**. Each *series* represents the data of a specific field from a particular data source.

2.2 Related Work

2.2.1 Semantic Caching. Traditional caching mechanisms[2, 4, 34] rely on exact key-based matching, which fails to identify intersections of results from similar queries. In contrast, semantic

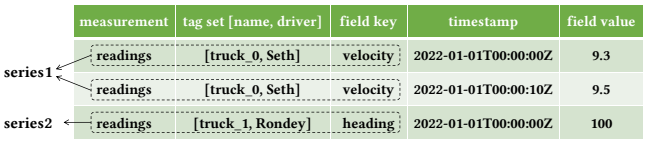


Figure 1: An example of points and series.

caching[31, 61] caches both query results and semantics, answering new queries by matching semantics, thereby fully reusing cached data. Semantic caching has been widely studied [24, 43, 72, 73, 76] and applied in various domains, such as client-server database [31, 61], mobile computing [50, 59, 60], web services [28, 29], Large Language Models [23, 32], and performance monitoring [77]. BSCache [77] is a lightweight semantic cache for cloud-based performance monitoring time-series systems. It focuses on the semantic relationships between metadata (*e.g., tags*) and data (*timestamps*, *field values*). However, it overlooks semantic relationships between queries, underutilizes semantic caching potential, and its memory-based design is cost-prohibitive for large-scale caching of massive time-series data. These limitations drive the design of STsCache.

2.2.2 Query Containment. The query containment problem [45, 48, 56, 62] is a fundamental issue in data management. It asks whether, for two queries Q_1 and Q_2, $Q_1(D)$ is contained within $Q_2(D)$ for any database D. Here, $Q_i(D)$ refers to the result of executing Q_i on database D. Chen *et al.* [26] proposed XCache, an XML semantic cache that determines query containment using regular expression-like pattern variables. Luo *et al.* [55] designed a semantic cache for web proxy, based on query containment for keyword-based and spatial queries. However, prior work on query containment is unsuitable for time-series semantic caching. Time-series queries, involving predicates, thresholds, aggregations, and time-ranges, differ significantly from other query types. This motivates formal definitions of semantic relationships in time-series queries.

2.2.3 Materialized View. *Materialized views* [25, 36, 69, 71] precompute store and query results to reduce real-time computation costs. They retain SQL definitions like regular views [33, 65] and can be periodically refreshed. MVs are used in some TSDBs (e.g., TimescaleDB's continuous aggregates [12]), but they rely on predefined query logic by DBAs and may add storage and refresh burdens to the database server. In contrast, semantic caching matches query semantics more flexibly without predefined logic and operates independently to avoid resource competition with the database server.

2.3 Time-series Caching

TSCache [53] builds time-series caches on flash memory. It uses two composite indexes to accelerate data retrieval. It uses a columnar slab structure for data, with key arrays and bit arrays to match queries to cache entries. It also employs a time-aware mechanism to manage cache replacement. However, as TSCache calculates the hash value of the query excluding the time-range predicate, it ignores the semantics, leading to low hit rates under practical workloads. MinMaxCache [57] provides an error-bounded, in-memory adaptive caching for visualization applications. The success of MinMaxCache in visualization applications has motivated us to design caching schemes for broader time-series data applications.

3 FORMAL DEFINITION OF SEMANTIC CACHING IN TSDBS

According to the elegant work on semantic cache in generic queries [31, 61], the meaning (referred to as semantic) of a query is interpreted as the *projected columns*, *predicates*, *etc.* involved in the query. Accordingly, semantic caching studies whether cached data can satisfy a new query by comparing the correlation between their semantics (*i.e.,* query meaning), *i.e.,* the *projected columns*, *predicates*, *etc.* In this regard, we have been following this definition and put it one step further, *i.e.,* formally defining time-series query semantics as well as exploring the semantic relationships between them. Based on well-known benchmarks [8, 14, 15, 17, 54] and related work [37, 39, 44, 47, 66, 70, 79], we identified that time-series query patterns typically include time-ranges, aggregations, threshold filtering, and tags. For clarity, we present the following query examples in an SQL-like language.

- **Q1:** Basic time-range query.

```
SELECT velocity, heading FROM readings
WHERE name='truck_0' AND time>='2022-01-01T09
    :00:00Z' AND time<'2022-01-01T12:00:00Z';
```

- **Q2-1:** Time-range query with tag grouping.

```
SELECT velocity, heading FROM readings
WHERE (name='truck_0' OR name='truck_1') AND
    time>='2022-01-01T09:00:00Z' AND time<'
    2022-01-01T12:00:00Z'
GROUP BY name;
```

- **Q2-2:** Time-range query with tag grouping.

```
SELECT velocity, heading FROM readings
WHERE (name='truck_0' OR name='truck_2') AND
    time>='2022-01-01T08:00:00Z' AND time<'
    2022-01-01T11:00:00Z'
GROUP BY name;
```

- **Q3:** Time-range query with field predicate.

```
SELECT velocity, heading FROM readings
WHERE name='truck_0' AND time>='2022-01-01T10
    :00:00Z' AND time<'2022-01-01T11:00:00Z'
    AND velocity>99;
```

- **Q4:** Time-range query with aggregation.

```
SELECT max(velocity) FROM readings
WHERE (name='truck_0' OR name='truck_1') AND
    time>='2022-01-01T11:00:00Z' AND time<'
    2022-01-01T13:00:00Z'
GROUP BY name, time(10m);
```

3.1 Query Semantics

Suppose that there is a TSDB instance, denoted as D, let M be a specific combination of values for *measurement* and *tag sets* within D, which can be used to denote one or more data sources. For example, in Figure 1, there are two Ms, *i.e.,* $M_1 = readings.(name = truck\_0, driver = Seth)$ and $M_2 = readings.(name = truck\_1, driver = Rondey)$. Then, we use F to denote the collection of field keys (*e.g., velocity*, *heading*) in D.

Next, let P be the field predicate, and let P_F be the field keys involved in P. Then, let $T = [ts_{begin}, ts_{end}]$ be a time-range, where ts denotes a timestamp and $ts_{begin} \leqslant ts_{end}$. For instance, consider query **Q3**, $P = velocity > 99$, $P_F = \{velocity\}$, T of **Q3** is $[2022-01-01T10:00:00Z, 2022-01-01T11:00:00Z]$.

Table 1: Notations.

Notations	Descriptions
q / $\hat{q}$	A cached query. / A new query.
D	A time-series database.
M / q_M	A combination of measurement and tag sets. / All Ms in q.
F / P_F / q_F	All field keys of D / P / q.
P / q_P	Field predicate. / P in q.
q_G	Aggregation function and time interval in q.
T / q_T	Time-range predicate. / T in q.
$Series(q, M)$	Semantics corresponding to each data source.
$Meta(q)$	Some $Series$ within a specific time-range.
$Metric(q_F, q_P, q_G)$	A collection of $Series$ with same q_F, q_P, q_G.

Given the above, we formally define *Semantic Meta* as follows.

Definition 1 (Semantic Meta). *Given a time-series query q, the semantics contained in it is defined as the **Semantic Meta** of q, denoted in the following form.*

$$Meta(q) =< q_M, q_F, q_P, q_G, q_T >,$$

where q_M refers to the corresponding set of M values defined in q, q_F are the field keys appearing in q, q_P and q_T are the field predicate and the time-range predicate of q, respectively. $q_G = \{Aggr(*), interval\}$, where $Aggr(*)$ can be any of {COUNT, MEAN, SUM, MAX, MIN} or kept empty; $interval$ can be empty or a user-specified time interval used in GROUP BY time() clause.

Example 1. *The semantic meta of **Q4** can be represented as follows.*

$$<q_M = \{readings.(name = truck\_0), readings.(name = truck\_1)\},$$
$$q_F = \{velocity\}, q_P = \varnothing, q_G = \{MAX, 10m\},$$
$$q_T = [2022{-}01{-}01T11{:}00{:}00Z, 2022{-}01{-}01T13{:}00{:}00Z] > .$$

Notably, q_M may contain multiple Ms (*e.g.,* Example 1), referring to different data sources. Therefore, to differentiate the meta between data sources in queries, we define *Semantic Series* as follows.

Definition 2 (Semantic Series). *Given a time-series query q, the **Semantic Series** identifies the semantics corresponding to each data source in q,* i.e., *a specific value of M ($M \in q_M$), can be formally represented as follows.*

$$Series(q, M) =< M, q_F, q_P, q_G > .$$

Obviously, each semantic meta corresponds to one or more semantic series within a specific time-range, *i.e.,*

$$Meta(q) =< \{Series(q, M_i) | M_i \in q_M\}, q_T > .$$

In the above equation, all $Series(q, M_i)$ share the same value of q_F, q_P and q_G, we further define *Semantic Metric* accordingly to denote such a collection of semantic series.

Definition 3 (Semantic Metric). *Given a particular set of values of q_F, q_P, and q_G, the collection of those semantic series $Series(q, M) =< M, q_F, q_P, q_G >$ is referred to as **Semantic Metric** of q_F, q_P, q_G, formally represented as follows.*

$$Metric(q_F, q_P, q_G) = \{Series(q, M) | Series(q, M) =< *, q_F, q_P, q_G >\}.$$

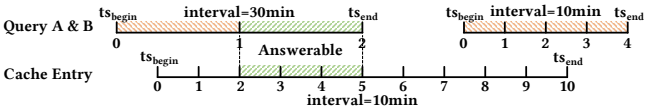


Figure 2: An illustration of Aggregated Answer.

3.2 Semantic Relationships between Queries

Given the above definitions, we are ready to explore the semantic relationship between $\hat{q}$ and a cached query q. Suppose that $\hat{q}_T \cap q_T \neq \varnothing$ and $M \in \hat{q}_M \cap q_M$, depending on whether and how the results corresponding to $Series(q, M)$, which have been cached, can be relied upon to answer $Series(\hat{q}, M)$, we can categorize the answer paths into three types: ❶ If $Series(\hat{q}, M) = Series(q, M)$, *i.e.,* they have identical field keys, predicates, then the results of $Series(q, M)$ can be directly employed to answer part of $Series(\hat{q}, M)$, subject to $\hat{q}_T \cap q_T \neq \varnothing$. ❷ Otherwise, if $Series(\hat{q}, M).\hat{q}_G = Series(q, M).q_G$, and the field keys of $\hat{q}$ are contained by that of q, with $\hat{q}_P$ contained by q_P, we consider that the results *w.r.t.* $Series(q, M)$ can be used to answer $Series(\hat{q}, M)$ through filtering. ❸ Finally, if $\hat{q}_P = q_P$, $\hat{q}_F$ is contained by q_F and they share the same aggregation types but differentiate in interval (or $q_G = \varnothing$), we check if the groups aggregated in $Series(\hat{q}, M)$ can align with that of $Series(q, M)$. If alignable, the results *w.r.t.* $Series(q, M)$ can be used to answer $Series(\hat{q}, M)$ by aggregating the cached groups. As shown in Figure 2, $\mathbf{Q_A}$ (with $\hat{q}_G.interval = 30min$) can be answered by cache results (with $q_G.interval = 10min$) as the second aggregation interval for $\mathbf{Q_A}$ (upper green) aligns exactly with three cached intervals (bottom green). In comparison, $\mathbf{Q_B}$ cannot be answered by the cache.

Accordingly, we formally define the answer paths as follows.

Definition 4 (Cache Answerable). *Given a pair of queries, $\hat{q}$ and q, subject to $M \in \hat{q}_M \cap q_M$, $\hat{q}_F \subseteq q_F$, $\hat{q}_T \cap q_T \neq \varnothing$, in the following cases $Series(q, M)$ can be relied upon to answer $Series(\hat{q}, M)$, referred to as $Series(q, M)$ is **answerable** to $Series(\hat{q}, M)$.*

- ***Direct Answer:** Results for $Series(q, M)$ can be directly used to answer the $Series(\hat{q}, M)$, if: $Series(\hat{q}, M) = Series(q, M)$;*
- ***Filtered Answer:** Results for $Series(q, M)$ needs to be filtered before it can be used to answer $Series(\hat{q}, M)$, if: $\hat{q}_P \Rightarrow q_P$ and $\hat{q}_{P_F} \subseteq q_F$ and $\hat{q}_G = q_G$;*
- ***Aggregated Answer:** Results for $Series(q, M)$ needs to undergo aggregation before it can be used to answer $Series(\hat{q}, M)$, if, $\hat{q}_P = q_P$, ($\hat{q}_G.Aggr(*) = q_G.Aggr(*)$ or $q_G.Aggr(*) = null$) and $\hat{q}_G.interval = k \cdot q_G.interval$ ($k \in \mathbb{N}^+$). Notably, it also requires that $\hat{q}_T.ts_{begin} + k_1\hat{q}_T.interval = q_T.ts_{begin} + k_2 q_T.interval$ and $\hat{q}_T.ts_{begin} + k_3\hat{q}_T.interval = q_T.ts_{begin} + k_4 q_T.interval$ ($k_1 < k_3$ and $k_2 < k_4$, $k_1, k_2, k_3, k_4 \in \mathbb{N}$).*

Definition 4 studied 'Whether', but not 'How', the results for a cached query q can be relied upon to answer a new query $\hat{q}$. We now move on and study 'How' $\hat{q}$ can be answered by q. We present two cases of cache hit, namely *full hit* and *partial hit.*

Definition 5 (Full hit and Partial hit). *Given a cached query q and a new query $\hat{q}$, we define two types of cache hits as follows:*

- ***Full Hit:** $\hat{q}$ can be fully answered by q, if:*
  - *$\forall Series(\hat{q}, M)$, $\exists Series(q, M)$, $Series(q, M)$ is answerable to $Series(\hat{q}, M)$ and $\hat{q}_T \subseteq q_T$.*

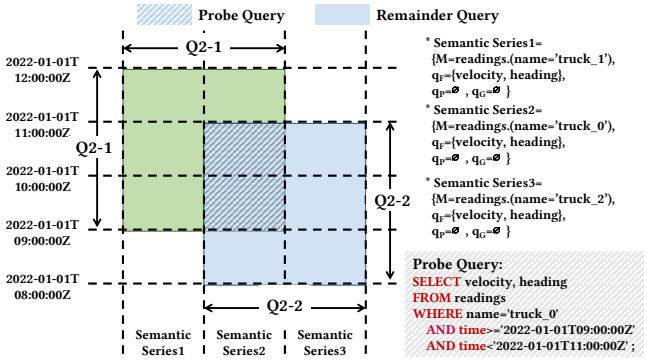


Figure 3: Query trimming for Q2-2 by Q2-1 cache.

- ***Partial Hit:*** *$\hat{q}$ can be partially answered by q, if:*
  - *$\exists Series(\hat{q}, M)$, $\exists Series(q, M)$ such that $Series(q, M)$ is answerable to $Series(\hat{q}, M)$ and $\hat{q}_T \cap q_T \neq \varnothing$.*

In the former case, all the results of $\hat{q}$ can be found from the cache *w.r.t.* q. Whenever the latter case happens, the answer to $\hat{q}$ can be decomposed into two parts: the cached results *w.r.t.* q that supply as a partial answer to $\hat{q}$ and the marginal records that belong to $\hat{q}$ but are not in the cache. We refer to the query task for each part as *probe query* and *remainder query*, respectively.

Example 2. *Suppose that **Q2-1** has been executed before, the semantics and query results have been cached correspondingly. Consider that a new query **Q2-2** is submitted, it is easy to find that **Q2-2** can be partially hit by **Q2-1** following Definition 5. Moreover, the cached answer of **Q2-1** does not need to undertake filter or aggregation, according to Definition 4. Therefore, **Q2-2** can be answered via a probe query and a remainder query (details in Figure 3).*

4 STSCACHE DESIGN

Based on the definitions above, we present STsCache, a semantic time-series caching scheme. Like Memcached [34], STsCache is a standalone system using a cache-aside architecture (see Figure 4). Clients first access STsCache. If there's no *full hit*, they query on TSDBs and store the query semantics and results in STsCache. The cache-aside architecture decouples STsCache from TSDBs, eliminating the need for cumbersome data loading logic. Developers can flexibly manage cache through the client programs we provide, which are modified from existing clients [13] with minimal changes.

4.1 Architecture Overview

As shown in Figure 5, STsCache consists of the following parts:
Semantic-based Interface. We have designed an interface based on query semantics to provide caching services for general TSDBs.
Data Management. The basic unit of data management in STsCache is the slab, which is a collection of chronologically ordered data points of the same *semantic series*. Slabs can be stored in memory and SSD to balance performance and memory overhead.
Semantic Index. A composite index composed of a graph and skip lists, which can store the semantic relationships of historical queries. Accessing cached data through the semantic index avoids a large number of expensive semantic matching processes.
Eviction. It uses *semantic value* to quantify slab contributions and prioritizes retaining high-value slabs in memory. When memory is full, it evicts a batch of low-value slabs to SSD.
Compaction Module. Deduplication insertion and lazy compaction are presented to improve the utilization rate of cache space.

4.2 Semantic-based Interface

In order to provide an efficient semantic caching service for general TSDBs while keeping the easy-to-use experience like key-value caches, we have designed a custom serialization method and an interface similar to key-value caching.

4.2.1 Customized Serialization and Deserialization. STsCache converts queried data points to binary and stores them contiguously (see Figure 6). Using common serialization methods like Protobuf and JSON embed metadata into binary data for parsing, which forces STsCache to deserialize and re-encode data points into contiguous binary format, adding overhead. Moreover, text-based serialization (*e.g.,* JSON) also increases data size for network transmission.

Therefore, we propose custom serialization and deserialization methods. For query $\hat{q}$, we sequentially encode each $Series(\hat{q}, M_i)$ in $Meta(\hat{q})$ (see Definition 2) and its data points into binary format. This enables STsCache to store data points directly, avoiding redundant deserialization and re-encoding. During query responses, STsCache sends results to the client in the same format, which the client converts into query results via custom deserialization.

4.2.2 API Design. Similar to TSCache [53], we also provide an interface with the time-range parameter. But differently, instead of hashing the query into a key, we introduce SemanticKey, which we keep as a standard string format that includes all the content of *semantic meta* except for the time-range, *i.e.,* $\{Series(\hat{q}, M_i) | M_i \in \hat{q}_M\}$. STsCache focuses solely on parsing SemanticKey, avoiding compatibility issues with diverse TSDB query formats.

Set(SemanticKey, BeginTime, EndTime, Value). Add the semantics and results of a query to STsCache. Value is a byte stream by serializing the query result. For empty queries, clients set Value to 'NULL'.

Get(SemanticKey, BeginTime, EndTime). Retrieve the query results from STsCache. The client receives binary data and converts it into query results through deserialization.

4.2.3 Client for STsCache. We implemented our designed API based on an advanced Memcached client [13]. Developers can use the modified client (*i.e.,* STsCache Client) to interact with STsCache. To simplify usage, we encapsulated the cache-aside architecture logic and custom serialization/deserialization into a client called STsCache-Aside Client. This client interacts with both STsCache and TSDBs and merges query results from both sources.

4.3 Slab-based Semantic Data Management

The basic data unit in STsCache is the slab. Each slab is a contiguous block of space in memory or SSD, used to store the data points associated with the same *semantic series* in chronological order. Due to the *append-only* and *write-once* characteristics of time-series data, STsCache does not need to consider cache consistency issues.

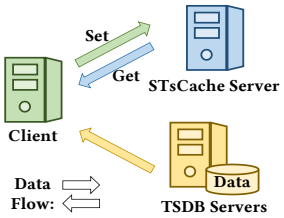


Figure 4: Data flow.

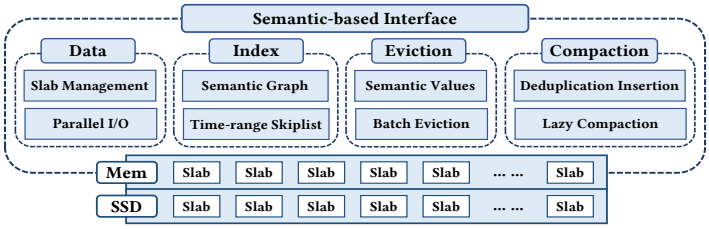


Figure 5: Architecture of STsCache.

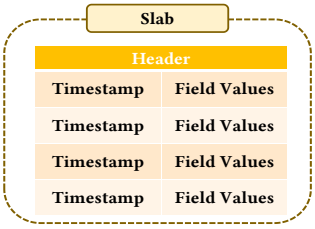


Figure 6: Structure of Slab.

This method of data storage brings several advantages. ❶ Since the data points within a slab are associated with the same *semantic series*, there is no need to store metadata for each data point within the slab to record their semantics. Minimal metadata for the entire slab suffices, enhancing storage space utilization. ❷ According to Definition 4, we can determine whether the *semantic series* of the slab is answerable to the new query $\hat{q}$, without the need to access each data point within the slab, thus improving cache access efficiency. ❸ Slab-based storage enables efficient sequential writes in 4KB multiples, which extends the lifespan of flash devices.

4.3.1 Slab Structure. As shown in Figure 6, a slab consists of two parts: *header* and *data*. The former stores a small amount of metadata. The latter stores data points associated with the same *semantic series* in chronological order. Unlike row or column storage, the slab stores all fields corresponding to the *semantic series* involved. Compared to row storage, it avoids the waste of row space when accessing only a subset of fields. Compared to column storage, it can prevent a large number of random I/O during read and write.

In terms of size, intuitively, a larger slab size can benefit more from the sequential I/O performance of flash devices. However, a larger capacity is more prone to causing space fragmentation. According to our experiments (see Section 5.4), overall performance is the best when the slab size is set to 16KB.

4.3.2 Parallel Slab I/O Operations on NVMe SSDs. The I/O overhead of SSDs is greater than that of memory and network [49], making it the bottleneck of the entire caching system. Batch reading and writing of slabs from SSDs offers the opportunity for multi-threaded parallel I/O. Therefore, we have equipped STsCache with a thread pool for parallel I/O on NVMe SSDs. High degrees of parallel random I/O can leverage the performance of NVMe SSDs [38, 75].

4.4 Semantic Index

For semantic caching, indices must also consider query semantics to accelerate matching. We propose an efficient semantic index comprising two parts (see Figure 7): a *Semantic Graph* and *Time-range based Skip Lists*, the former is used to store and manage *semantic series* and their semantic relationships (*i.e.,* answer path in Definition 4); and the latter accelerates data retrieval within a given time-range.

4.4.1 Semantic Graph. We present a graph-based index to record the answer path between $Series(\hat{q}, M)$, *i.e.,* slabs.

Semantic Node. Intuitively, it is rational to denote each cached $Series(\hat{q}, M)$ using a node, and connect nodes based on the answer path. However, it may result in a large number of nodes and accordingly improve the search complexity within the graph index. Therefore, to limit the scale of the graph index, we select to correlate each Semantic Node to a *semantic metric*, but not a single *semantic series*. As the *semantic series* within a $Metric(\hat{q}_F, \hat{q}_P, \hat{q}_G)$ correlates with multiple M, we introduce a Mapping List to map each specific value of M to a Time-based Skip List.

Semantic Edge. As the answer path in Definition 4 is defined based on the relationship between $\hat{q}_P, \hat{q}_F, \hat{q}_G$ (assuming identical $\hat{q}_M$), each value of which corresponds to a semantic node, it is applicable to connect pairs of semantic nodes according to the answer path accordingly. The weights of edges store the corresponding answer path in Definition 4. Notably, a Semantic Edge $u \longrightarrow v$ indicates that the semantic series in u is answerable to the semantic series in v that has the same M.

4.4.2 Time-range based Skip List. We design a time-range based skip list for efficient data retrieval within a time-range. Each node has a non-overlapping *time-range key* T, and is associated with a list of slabs (Sids in Figure 7) falling into T. In particular, it is possible that the time-range for a pair of nodes becomes overlapped due to data point insertion (*resp.,* deletion), in that case the corresponding nodes in the skip list can be merged (*resp.,* split). Intuitively, we can adopt the time-range of each slab in the *time-range keys* in the skip list. However, this can never cache the time-range of empty queries and leads to cache penetration. To address that, we select to adopt a query-driven index construction, where time-range of the query semantics is used as the *time-range keys* and store the time-range of the data points in the corresponding slab in the nodes of the slab list. As shown in Figure 7, the time-range node $[t_6, t_{11}]$ is associated with a slab list that contains $[t_7, t_8]$ and $[t_9, t_{10}]$, each of which corresponds to a single slab, where $t_6 \leqslant t_7 \leqslant t_8 < t_9 \leqslant t_{10} \leqslant t_{11}$.

This can bring benefits in the following aspects: ❶ Rapid determination of cache hits. It can efficiently find nodes where the *time-range keys* overlap with the time-range of the current query. ❷ More granular search. A node may be associated with many slabs. We further refine the slabs we want to search based on the time-ranges in the slab list. ❸ Prevent cache penetration. We store the time-ranges of empty queries as *time-range keys*. If query q matches the time-range keys in the index but no slab is found, STsCache will return a cache hit directly, effectively handling cache penetration.

4.4.3 Operations Based on Semantic Index. We implement a pair of SetByIndex and GetByIndex operations by the semantic index.

Set by Index. Algorithm 1 illustrates the process of Set. It parses the *SemanticKey* to obtain a list of semantic series, and then stores the semantics and data into the slab within the cache. If there is not a semantic node containing the series, we create one accordingly and connect it with other nodes based on the answer path.

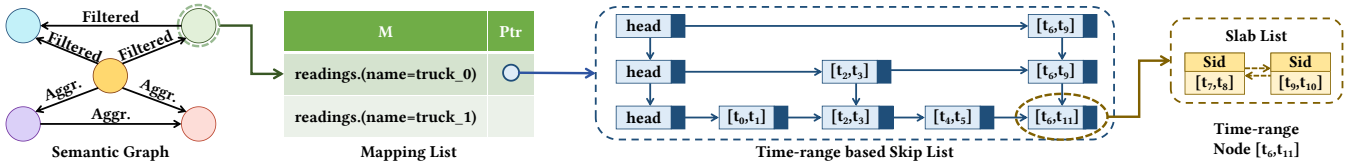
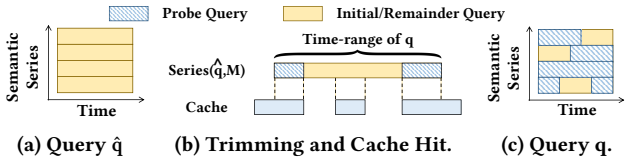


Figure 7: Semantic index.

Algorithm 1: SetByIndex

Input: I: Semantic index, K: SemanticKey from Set, D: Data to be inserted, T: Time-range of D
Output: I': I after set

```
Function SetByIndex(&I,K,D,T):
    SSeriesList = ParseSkey(I) ; SGraph = GetSGraph(I) ;
    SNode = FindSNode(Graph,SSeriesList) ;
    if SNode == null then
        SNode = CreateSNode(series) ;
        Edges = FindSRelationship(Graph,SNode) ;
        Append(&G,Edges,SNode)
    foreach series ∈ SSeriesList do
        TSkipList = FindTSkipList(SNode,series) ;
        if TSkipList == null then
            TSkipList = CreateTSkipList(series) ;
            SetMap(&SNode, series,TSkipList) ;
        DeduplicationInertion(D,T,&TSkipList) ;
```



(a) Query $\hat{q}$ (b) Trimming and Cache Hit. (c) Query q.

Figure 8: An illustration of Query Trimming.

Query Trimming. According to Definition 2, the semantics of query $\hat{q}$, *i.e.*, $Meta(\hat{q})$, composed of $\{Series(\hat{q}, M_i)\}$ and $\hat{q}_T$, can be plotted as a rectangle on the semantic series and time-range space (Figure 8a). We perform query trimming based on semantic series. We only trim the query when the cache covers either $\hat{q}_T.ts_{begin}$ or $\hat{q}_T.ts_{end}$, allocating the cache-hit parts to the *probe query* and the cache-miss parts to the *remainder query* (Figure 8b). Each semantic series in *remainder query* is linked to a continuous time-range, simplifying the client's result merging process (Figure 8c).

Get by Index. Algorithm 2 illustrates the Get process. We trim the query and retrieve data by *semantic series* and time-range. If the query is not met, we traverse the incoming edges of the *Semantic Node* to retrieve data for the remainder time-range, *i.e.*, *RemainT*. The time complexity of Algorithm 2 is $O\left(\sum_{i=1}^{n_v}(1+\log(N_i))\right)$, where n_v denotes the number of semantic nodes traversed for cache access and N_i represents the number of slabs for the corresponding skip list index. Without index-based access to the cache, the complexity escalates to $O(N \cdot \sigma)$, where N is the number of slabs ($N \geq \sum_{i=1}^{n_v} N_i \gg n_v$) and σ refers to the time required for one semantic match strictly following Definition 4 and Definition 5. Algorithm 2 reduces the overhead of large-capacity cache access by efficiently replacing costly and frequent semantic matches with graph traversal and skip list searches.

Algorithm 2: GetByIndex

Input: I: Semantic index, K: SemanticKey from get, T: Time-range of query
Output: D: Data retrieved from cache

```
Function GetByIndex(I,K,T):
    SSeriesList = ParseSkey(I) ; SGraph = GetSGraph(I) ;
    SNode = FindSNode(Graph,SSeriesList) ; D = [] ;
    if SNode ≠ null then
        foreach series ∈ SSeriesList do
            TSkipList = FindTSkipList(SNode,series) ;
            RemainT = T ;
            if TSkipList ≠ null then
                // trim query and get probe query's data.
                RemainT = GetData(TSkipList, T, &D) ;
            if RemainT ≠ null then
                foreach edge ∈ GetInEdges(SNode) do
                    // trim query and get probe query's data.
                    GetDatabyEdge(edge, &RemainT, &D) ;
                    if RemainT == null then
                        break;
    Return D ;
```

4.5 Batch Eviction Based on Semantic Value

When cache space is full, retaining valuable content for future queries is key to maintaining high hit rates. In hybrid storage caches, this involves keeping valuable slabs and hotter data in memory.

4.5.1 Semantic Value. To quantify the eviction priority of a slab, we propose using the total read/write data volume of the slab, referred to as *semantic value*. In high-frequency read-write caching systems, real-time sorting of all *semantic values* is costly, so we sort them only during eviction. We create a *semantic value array* to track semantic values of all slabs. On slab eviction, the value resets to zero. On each read/write for a slab, its semantic value increases.

4.5.2 Aging. We address slab eviction misguidance by old hot data using an aging strategy that periodically halves semantic values via bitwise operations. To reduce traversal costs in large caches, we hide the aging traversal in the eviction process. We introduce a factor to denote the weight of current operations: each read/write shifts the semantic value left by the factor bits, and during batch replacement, the value is shifted right by the factor bits to age it. The factor is incremented regularly and reset to 0 at traversal end, giving higher weight to recent operations.

Algorithm 3: Batch Eviction Based on Semantic Values

Input: S: Semantic value array, L: List of used slab IDs, F: List of free slab IDs, E: List of evictable slab IDs, K: batch size
Output: S', L', E', F': S, L, E, F after eviction

```
Function BatchEviction(&S,&L,&E,&F,K):
    // Two min-heaps sorted by S[id].
    MHead = InitMinHeap() ; DHead = InitMinHeap() ;
    foreach sid ∈ L do
        S[sid] » factor ;
        if IsMem (sid) then
            PushHead(&MHead, sid) ;
        else
            PushHead(&DHead, sid) ;
    Num = 0 ; WriteList = InitList() ;
    while !IsEmpty(MHeap,DHeap) and Num++ < K do
        MTop = GetTop(MHead) ; DTop = GetTop(DHead) ;
        if S[DTop] ⩽ S[MTop] then
            DeleteSlab(&L, DTop) ; PopHead(&DHead) ;
            InsertList(&F, DTop) ;
        else
            PopHead(&MHead) ; InsertList(&E, MTop) ;
    for i = 1 → K − GetSize(E) do
        MTop = GetTop(MHead) ; PopHead(&MHead) ;
        InsertList(&WriteList,Mtop) ;
    // Write to SSD slabs and free memory slabs.
    BatchWriteSSD(WriteList, &L, &F) ;
```

4.5.3 Batch Evicton. We propose a batch eviction algorithm based on semantic values (see Algorithm 3). First, we select k slabs with the lowest semantic values, evicting those in SSD directly and placing those in memory into the eviction list E. Slabs on the list E are ready for quick eviction and insertion. Next, we search for **(k - GetSize(E))** additional slabs in the memory slab list and batch-write them into the SSD. This approach has three benefits: ❶ Batch eviction spreads the overhead of sorting semantic values. ❷ It retains k free or evictable slabs in memory for efficient cache insertion. In our experiments, k is set to $\frac{1}{16}$ of the number of slabs in memory to achieve optimal performance (see Section 5.5). ❸ Parallel writing of slabs takes full advantage of the hardware bandwidth.

4.6 Compaction Module

Although managing data in slabs leverages SSD sequential I/O and reduces management costs, it causes cache space wastage due to two main issues. ❶ Insertion of duplicate data. The results of different queries may overlap; hence, STsCache faces the issue of inserting duplicate data. ❷ Space fragmentation within slabs. STsCache uses slabs in memory to store incoming data for rapid processing, rather than strictly using space in a time-sequential manner from slabs that are not fully utilized. This leads to space fragmentation in many slabs. Therefore, we propose a *deduplication insertion algorithm* and a *lazy compaction algorithm* to address the two issues above.

4.6.1 Time-aware Deduplication Insertion. Inserting duplicate data into STsCache causes extra I/O, storage costs, and read amplification. We employ a time-aware mechanism to avoid the insertion of duplicate data into STsCache (see Algorithm 4). We identify all **InsertPoints**, which are composed of the time-range of non-duplicate data and the ID of the slab that can be inserted (see Figure 9). To ensure efficiency, the target slab must be in memory. We then traverse the **InsertPointList**, locate non-duplicate data intervals by time-range, and insert them into specified or new slabs. Since data points are chronological, binary search enables quick location.

Algorithm 4: Time-aware Deduplication Insertion

Input: D: Data to be inserted, T: Time-range of D, L: Time-range based skip list
Output: L': L after data insertion

```
Function DeduplicationInsertion(D,T,&L):
    InsertPointList = FindInsertPoint(L,T) ;
    foreach point ∈ InsertPointList do
        SubData = GetSubData(D,point.TimeRange) ;
        InsertOver = false ;
        if point.Sid ≠ null then
            InsertOver = Insert(point.Sid,SubData) ;
        while !InsertOver do
            Sid = CreateSlab() ;
            InsertOver = Insert(Sid,SubData) ;
            UpdateSkipList(&L,point.TimeRange,Sid) ;
```

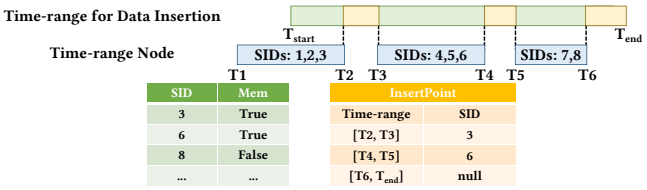


Figure 9: An example of Find InsertPoint.

4.6.2 Lazy Compaction Based on Time-range Node. To enhance cache space utilization and leverage SSDs' sequential I/O performance, we have designed a compaction mechanism that compacts multiple slabs associated with a time-range skip list node, storing logically adjacent data in contiguous space. Thanks to Algorithm 4, the compaction does not need to consider the deletion of duplicate data. However, compaction may still be costly, especially when slabs on SSD require extensive I/O. Therefore, we designed a lazy compaction mechanism that is triggered when inserting new data increases the number of slabs for a Time-range Node and at least one slab can be reduced through compaction.

5 EXPERIMENT

5.1 Implementation

We have implemented a prototype system of STsCache on Fatcache [9]. In our prototype, we compactly store data in restructured slabs and use a semantic index based on graphs and skip lists to accelerate cached data access. Unlike Fatcache's FIFO policy, we have implemented a batch eviction algorithm based on semantic values. We have integrated aggregation and filtered operators into the prototype, enabling it to answer new queries by processing the cached data. To provide caching services for general TSDBs, we have implemented Get and Set interfaces for STsCache, similar to

Table 2: Statistics of two TSBS datasets: Sample interval=10s, Duration (UTC): 2022.01.01T00:00:00Z ~2022.12.31T23:59:59Z.

Name	#fields	Scale	#devices	#data points	Size (GB)
IoT	16	Small	100	5,045,760,000	42.29
		Medium	500	25,228,800,000	211.47
		Large	1,000	50,457,600,000	422.93
DevOps	10	Small	100	3,153,600,000	25.85
		Medium	500	15,768,000,000	129.23
		Large	1,000	31,536,000,000	258.46

key-value caching systems, and implemented a client for STsCache based on gomemcache [13]. We have modified the Golang clients of InfluxDB and TimescaleDB to access STsCache and database services, as well as merge the query results. Our implementation for all the above results in about 10000 lines of C/C++/Golang code.

5.2 Experimental Setting

To evaluate STsCache, we established a testing platform and benchmarked STsCache comprehensively against baseline systems.

Setup. We use three DELL T7920 workstations as TSDB server, cache server, and client server. Each has a 2.9 GHz 16-core Intel Xeon Gold 6226R CPU, 128GB DDR5 RAM, 1TB SSD, 16TB HDD, and runs 64-bit Ubuntu 20.04 LTS. The dataset on the TSDB servers is deployed on the HDD. On the cache server, all SSD I/O operations bypass the OS page cache, executed in 'direct_io' mode.

Baselines. We will compare our approach with: ❶ TSDB (InfluxDB 1.8.10 [10] and TimescaleDB 2.6.0 [12]), ❷ TSCache [53], ❸ BSCache [77]. Notably, as BSCache in [77] is an internal system component, in order to conduct a comparison as a standalone caching service, we strictly follow [77] and implement it on Fatcache [9].

Datasets. we used two use cases from TSBS [14] for benchmarks.

- **IoT (Internet of Things).** This use case simulates diagnostic data and metrics streaming from a fleet of trucks.
- **DevOps (CPU-only).** This use case focuses only on CPU metrics, generating 10 metrics per read.

We set up three datasets of different scales for each case, with the data time span ranging from January 2022 to December 2022. More details on the datasets can be found in Table 2.

Workloads. We adopt the query patterns from TSBS [14], a well-known benchmark, which includes aggregations, threshold filtering, and tag grouping, *etc.* Additionally, referring to the YCSB-TS [8], we generate a reasonable time-range [time_begin,time_end] for queries based on the Latest distribution, the length of the query time-range following a Zipfian distribution between 3 hours and 1 month. In the IoT use case, average request sizes are 124.75 KB, 620.53 KB, and 1.21 MB for small, medium, and large datasets. In the DevOps use case, they are 82.34 KB, 413.96 KB, and 819.69 KB.

Metrics. We evaluate system performance based on three metrics: throughput, latency, and hit ratio. The unit of throughput is the number of queries per second, *i.e., queries/s.* End-to-end latency refers to the time from when the query request is sent until the client receives the query result object, with the unit *milliseconds (ms).* The hit ratio is classified into full hit ratio and partial hit ratio to measure the hit ratios for the two cases specified in Definition 5.

In our experiments, we enabled 64 client threads, which can fully utilize the system without overloading the server. Cache capacity was set to 4% of the dataset volume with a memory/disk ratio of 1:3. For each experiment, we regenerate and execute the workload five times and take the average metrics.

5.3 Overall Comparison

We set up six systems based on the baselines mentioned in Section 5.2 for comparison: ① *Single-TSDB.* A single TSDB server with data stored on HDD. ② *Single-TSDB-SSD.* A single TSDB server with data stored on SSD. ③ *Dual-TSDB.* As STsCache employs an extra server, for a fair comparison *w.r.t.* #servers, we construct a system with two TSDB servers, each of which stores a replica of the full datasets on SSD. The client distributes the query load across the two database servers in a round-robin manner. ④ *STsCache.* STsCache is deployed on the cache server, along with a TSDB server. Whenever there is not a *full hit*, the query results as well as the semantics *w.r.t.* the query are then stored by the cache server. ⑤ *BSCache* and ⑥ *TSCache.* Each is deployed as a cache server, with other settings identical to ④. Notably, TSCache defaults to only 128 query keys in its slab and employs a hashing-based matching mechanism, making it unsuitable for benchmarks. Thus, we separately configured a simplified workload for comparison with TSCache.

Overall Performance. Figure 10 illustrates the overall performance of these systems in all use cases and data scales. STsCache, *i.e.,* System④, performs best in all conditions. Compared to a single TSDB server, STsCache boosts throughput by 4.8-10.8× and cuts latency by 84.1%-92.4%. Compared to dual TSDB servers, it increases throughput by 1.8-4.0× and reduces latency by 65.6%-83.5%. These results show STsCache can provide excellent caching for TSDBs.

As shown in Figure 10 and Figure 11, STsCache outperforms existing semantic time-series caching schemes, achieving 1.5-3.4× higher throughput, 59.4%-77.1% lower latency, and 22.5%-38.8% higher hit ratios than BSCache. This superiority stems from two key factors. First, STsCache exploits semantic relationships between queries to enhance result reuse and employs a semantic index for efficient semantic matching. In contrast, BSCache overlooks these semantic relationships, diminishing result reuse. Second, BSCache is designed as a lightweight memory caching component, but its scheme is not suitable for large-capacity caching on hybrid storage. For example, BSCache's eviction algorithm requires calculating the distances between each storage unit and recent queries, which results in significant overhead in large-capacity cache systems.

SSD vs. HDD. We compared Single-TSDB to Single-TSDB-SSD to assess SSDs' impact on TSDB performance. Figure 10 shows minimal performance gains from SSDs due to CPU-bound operations like aggregation, filtering, and data decompression. STsCache, optimized for NVMe SSDs, caches query results and semantics to reduce redundant calculations. Unlike STsCache, InfluxDB and TimescaleDB are designed for general storage devices and lack specific optimizations for NVMe SSDs.

For ease of discussion and space limit, by default we shall report the results of the medium-sized dataset in the rest of this section.

STsCache vs. TSCache. As the default size of the key array in TSCache is 128, we need to simplify the workload by limiting query parameters such as *tag* combinations to a maximum of 128 keys (after hashing). For fairness, we set the TSCache's capacity to 4% of the data set, with a 1:3 memory/disk ratio. Figure 12 shows that

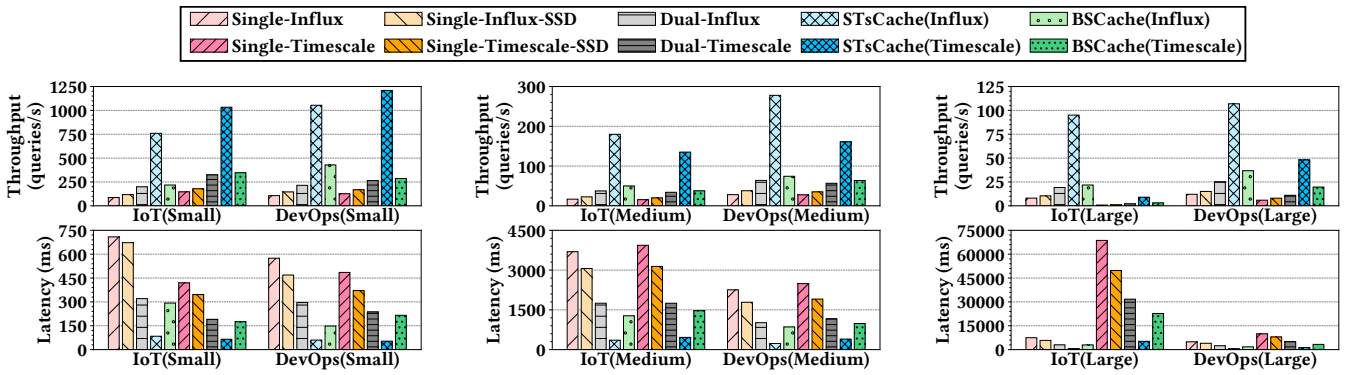


Figure 10: Throughput and latency of different systems on datasets of various scales.

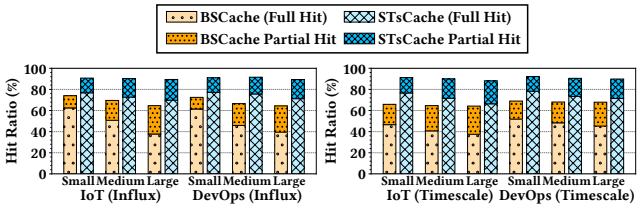


Figure 11: Hit ratio comparison for BSCache and STsCache.

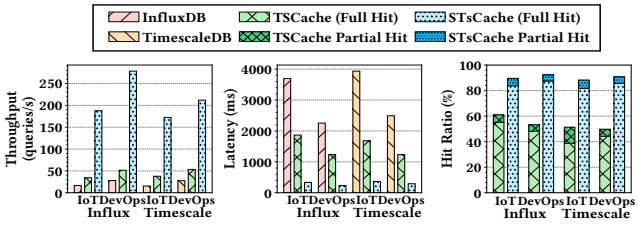


Figure 12: Comparison between TSCache and STsCache.

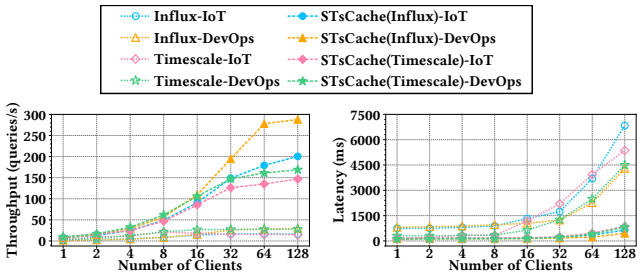


Figure 13: Effect of number of clients.

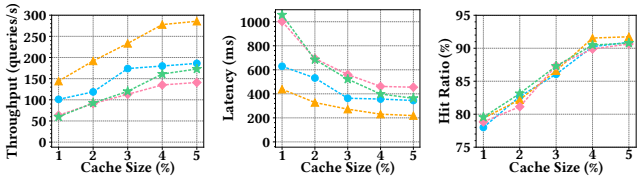


Figure 14: Effect of different cache sizes.

STsCache outperforms TSCache with 2.9-4.5× higher throughput, 76.4%-81.9% lower latency, and 46.9%-82.4% higher hit ratio.

STsCache outperforms TSCache for three main reasons. First, STsCache retrieves data more efficiently using the semantic index proposed in Section 4.4, while TSCache filters data through key and bit arrays. Second, STsCache has higher space utilization, storing more data points with only 8 bytes of metadata per slab, compared to TSCache's additional storage for key and bit arrays, which occupy over 50% of the slab space. Third, STsCache better reuses historical query data by integrating aggregation and filter operators, increasing cache hit opportunities.

Scalability. We compared scalability by increasing the number of clients. As shown in Figure 13, the throughput of STsCache also increases with more clients. However, the throughput of TimescaleDB (*resp.,* InfluxDB) reached a bottleneck when the number of clients increased to 16 (*resp.,* 32). Compared to a single client, STsCache achieved 16.7-36.8× higher throughput and only 3.4-7.4× higher latency with 128 clients, showing better scalability due to its efficient indexing and SSD-based parallel I/O.

Cache Size. To explore the effect of cache size, we vary cache sizes from 1% to 5% of the dataset size. As shown in Figure 14, STsCache's performance improves with increasing cache size. At 5% cache size, STsCache achieved 0.9-1.9× higher throughput, 45.2%-65.7% lower latency, and a 14.12%-16.5% higher hit ratio. Even at 1% cache size, STsCache still outperformed TSDBs.

Semantic Overlap. By preloading data into STsCache, we created varying levels of semantic overlap and conducted two experiments using InfluxDB: ❶ We compared STsCache with competitors on a medium-scale DevOps dataset with varying semantic overlap. ❷ We studied the impact of query size and semantic overlap on STsCache performance. As shown in Figure 15, STsCache achieves the lowest latency. Even with zero-overlap, STsCache incurs minimal latency increase from cache misses. Figure 16 shows that STsCache reduces latency with higher semantic overlap, especially for larger queries. For example, at a 4MB request size, 20% overlap reduces latency by 24.3%. At a 16KB request size, STsCache shows significant latency reduction only with full hits, as partial hits incur network overhead that offsets its performance benefits.

5.4 Slab Management

In this section, we explore the impact of the following parameters:

Slab Size. As shown in Figure 17, the cache performance improves as the slab size decreases from 1024KB to 16KB. When slab

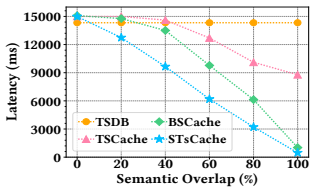


Figure 15: Latency under different semantic overlaps.

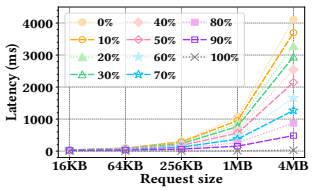


Figure 16: Latency with different request sizes and overlaps.

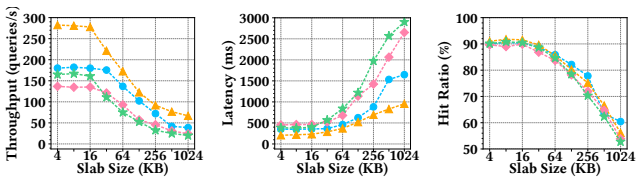


Figure 17: Effect of slab size.

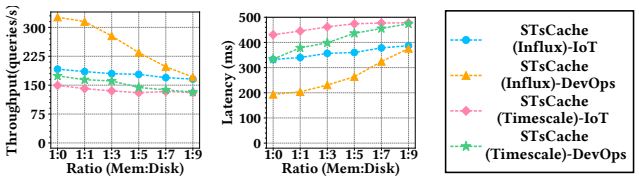


Figure 18: Effect of memory/disk ratio.

size decreases to 16KB, cache performance stabilizes. Given higher overhead for smaller slabs, *e.g.,* more space for metadata and larger arrays for semantic values, we set slab size to 16KB in experiments.

Memory/Disk Ratio. Figure 18 shows performance drops with higher disk storage proportions. To balance memory overhead and cache performance, we have chosen a memory/disk ratio of 1:3. Compared to a purely memory-based setup, throughput only decreases by 6.1%-14.9%, and the latency only increases by 7.2%-19.3%, while memory overhead is reduced by 75%.

5.5 Cache Replacement Policy

Effectiveness. To evaluate the batch eviction strategy, we extra implement FIFO and LRU in STsCache for comparison. Figure 19 shows that the batch eviction outperforms both FIFO and LRU. STsCache achieves 1.2-2.1× the throughput of FIFO and 1.1-1.5× that of LRU. Its average latency is 48.4%-83.6% of FIFO and 67.9%-89.1% of LRU. STsCache's hit ratio is slightly higher or on par with LRU, and significantly higher than that of FIFO. STsCache's full hit ratio is 20.9%-59.7% higher and its total hit ratio is 6.1%-10.7% higher than FIFO. FIFO evicts the oldest slab, ignoring its hotness, making it significantly worse than STsCache and LRU. LRU evicts the least recently accessed slabs, storing recent ones as hot data in memory, which is better than FIFO but still doesn't effectively distinguish between hot and hotter data. STsCache, however, quantifies slab hotness as semantic values, keeping hotter data in memory and hot data on SSD, thus boosting system performance.

Batch Size. In our setup, the batch size is set to $\frac{1}{16}$ of the slabs in memory. This choice, as Figure 20 shows, performs the best. A smaller batch size increases the cost of frequent evictions, which involve traversing and sorting data. A larger batch size can cause

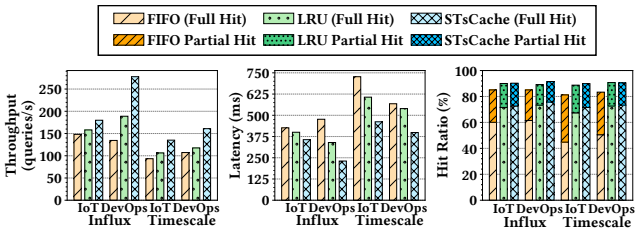


Figure 19: Effect of replacement policy.

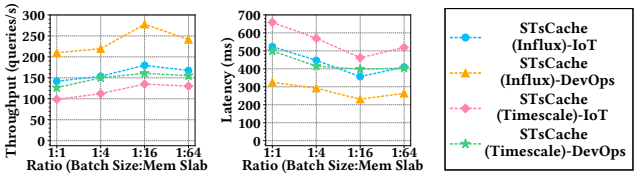


Figure 20: Effect of batch size.

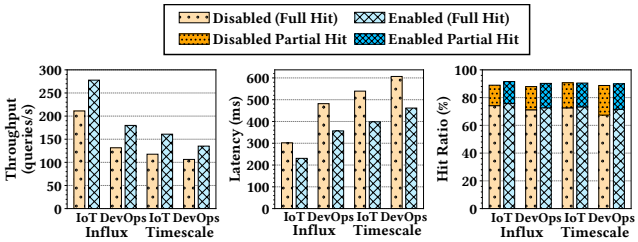


Figure 21: Effect of deduplication insertion.

delays due to a large amount of random I/Os on SSDs. It indicates that $\frac{1}{16}$ setting strikes a balance between eviction and I/O costs.

5.6 Compaction Module

Time-aware Deduplication Insertion. To evaluate the effectiveness of the time-aware deduplication insertion algorithm, we compared the performance of STsCache with and without deduplication insertion. As shown in Figure 21, STsCache with deduplication insertion shows improved throughput and latency, with a 27.1%-36.8% increase in throughput and a 23.6%-25.9% reduction in latency. Both use lazy compaction, so the hit ratio is similar. However, STsCache without deduplication insertion incurs extra costs for removing duplicates during compaction, leading to worse performance.

Lazy Compatcion. To evaluate the effectiveness of the lazy compaction mechanism, we compared the performance of STsCache without compaction, STsCache with real-time compaction, and STsCache with the lazy compaction, respectively.

Figure 22 shows that STsCache with lazy compaction performs best. In particular, real-time compaction can increase throughput by 8.6% -48.4%, reduce latency by 5.1% -33.2%, and improve the total hit ratio 4.5%-8.6%. These positive impacts of compaction have two main reasons. ❶ It reduces slab fragmentation, letting the cache store more data and raising the hit ratio. ❷ It compacts data storage, reducing random I/O. Compared to real-time compaction, the lazy compaction mechanism can further increase throughput by 3.2%-15.9% and reduce latency by 4.1%-13.7%. This is because the lazy compaction mechanism avoid frequent compaction overhead.

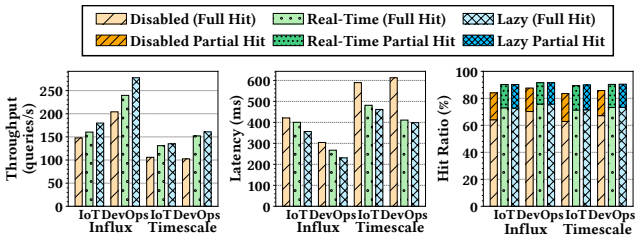


Figure 22: Effect of lazy compaction.

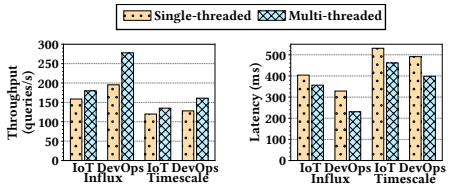


Figure 23: Performance under multi-threading.

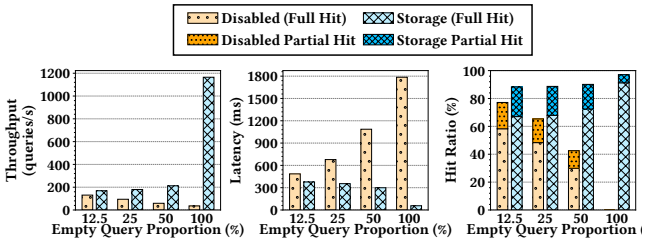


Figure 24: Performance under cache penetration.

5.7 Alleviate Performance Bottlenecks

Parallel Slab I/O Operations. Utilizing the parallel I/O bandwidth of NVMe SSDs can mitigate performance bottlenecks due to I/O overhead. To explore the effectiveness of parallel I/O, we compared the performance of single-threaded STsCache and multi-threaded STsCache. As shown in Figure 23, the multi-threaded parallel I/O has boosted STsCache throughput by 12.6% to 42.2% and reduced latency by 11.7% to 29.8%. This is because SSD I/O overhead is a bottleneck for STsCache, and sufficient parallelism significantly improves random I/O performance on NVMe SSDs.

Avoid Cache Penetration. STsCache stores empty query semantics in the index to prevent cache penetration. To verify that, we set up four workloads with empty query proportions of 12.5%, 25%, 50%, and 100% to simulate cache penetration. We refer to STsCache with and without empty query semantic storage as *STsCache-Storage* and *STsCache-Disabled*, respectively, and report the results in InfluxDB under the IoT use case. As shown in Figure 24, STsCache-Storage performs the best in all workloads. The performance gap widens as the empty query proportion increases. Especially in a 1:1 workload, storing empty query semantics enables STsCache to increase throughput by 31.5×, reduce latency by 96.8%, and increase the hit ratio by 97.1 percentage points. The results show that STsCache can effectively prevent cache penetration.

5.8 Production Environment Evaluation

We evaluated STsCache in the following two production environments:

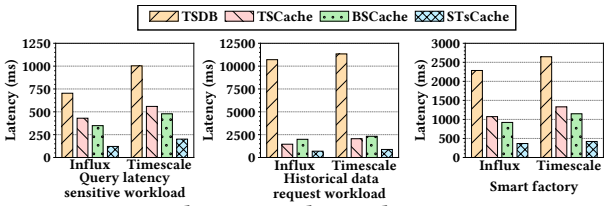


Figure 25: Query latency in the production environments.

① **DevOps (Monitoring and Analysis):** We **followed the production environment of BSCache [77]**. We deployed 100 containers with Node Exporter [20] using Docker [19]. Prometheus collects metrics from these 100 containers every 5 seconds and writes them to TSDBs. **We adopt the same two different workloads as [77]. WL.1** *Query latency-sensitive workload*, requesting the latest 10-minute and 5-minute data from an hour ago for ten key metrics (e.g., cpu_avg) for monitoring systems (*e.g.,* Grafana [11]). **WL.2** *Historical data request workload*, querying 12-hours data for half of time-series in the last 72-hours for analysis and prediction[35, 42]. Both queries run every minute.

② **Smart Factory:** We collected data from 200 devices (*i.e.,* 100 machine tools and 100 electricity meters) at a 15-second interval and wrote it into TSDBs. Smart factory applications require real-time monitoring and regular analysis of device data. **WL.3** *Smart factory workload* includes real-time queries for the latest 1 hour of data every 5 minutes and daily queries for the past week's data.

Experiments start after 1-week of data collection and run for 48-hours. Cache capacity is 512MB with a memory/disk ratio of 1:3. We integrated the client of STsCache into production applications (*e.g.,* Grafana [11]) to compare the end-to-end query latency against baselines. Figure 25 shows STsCache performs the best. Specifically, in **WL.1**, STsCache reduced latency by up to 82.8%, 71.8%, and 65.3% compared to TSDBs, TSCache, and BSCache, respectively. In **WL.2**, reductions reached 93.5%, 57.2%, and 65.5% compared to TSDBs, TSCache, and BSCache. In **WL.3**, reductions were 84.1%, 68.3%, and 63.3% compared to TSDBs, TSCache, and BSCache. For **WL.1** and **WL.3**, STsCache's gain over TSDB was lower than in **WL.2**, due to frequent queries for new data, which required more time to fetch from TSDBs. Overall, STsCache effectively offloads query pressure and reduces latency for TSDBs in production environments.

6 CONCLUSION

In this paper, we propose the formal definition of semantic time-series caching and present STsCache, a semantic caching system for hybrid memory and NVMe SSD storage. We have designed a series of components for STsCache, including slab-based cache unit, semantic index, semantic value-driven batch eviction, deduplication insertion, and lazy compaction. Exhaustive experiments demonstrated that STsCache can effectively enhance the efficiency of time-series queries compared to a series of baselines.

ACKNOWLEDGMENTS

This work was supported by the Special Task Project of the Ministry of Industry and Information Technology of China (No. ZTZB-23-990-024), and the National Natural Science Foundation of China (No. 62272369, 62302370).

REFERENCES

[1] 2008. Air Quality Dataset. https://archive.ics.uci.edu/dataset/360/air+quality
[2] 2009. Redis. https://redis.io/
[3] 2011. OpenTSDB. http://opentsdb.net/
[4] 2013. etcd. https://etcd.io/
[5] 2014. QuestDB. https://questdb.io/
[6] 2015. Prometheus. https://prometheus.io/
[7] 2018. Apache IoTDB. https://iotdb.apache.org/
[8] 2019. Yahoo Cloud Server Benchmark for Time Series. https://github.com/TSDBBench/YCSB-TS
[9] 2021. Fatcache: Memcache on SSD. https://github.com/twitter/fatcache
[10] 2021. InfluxDB. https://www.influxdata.com/
[11] 2022. Grafana: The open observability platform. https://grafana.com/
[12] 2022. TimescaleDB. https://www.timescale.com/
[13] 2023. Gomemcache: A memcache client library for the Go programming language. https://github.com/bradfitz/gomemcache
[14] 2023. Time Series Benchmark Suite: A tool for comparing and evaluating databases for time series data. https://github.com/timescale/tsbs
[15] 2024. ClickBench: A Benchmark For Analytical Databases. https://github.com/ClickHouse/ClickBench
[16] 2024. DB-Engines Ranking of Time Series DBMS. https://db-engines.com/en/ranking/time+series+dbms
[17] 2024. InfluxDB-comparisons: Benchmark suite for InfluxDB against other databases and time series solutions. https://github.com/influxdata/influxdb-comparisons
[18] 2024. Time Series Database - Ranking. https://ossinsight.io/collections/time-series-database/
[19] 2025. Docker. https://www.docker.com/
[20] 2025. Node exporter: Exporter for machine metrics. https://github.com/prometheus/node_exporter
[21] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L Wiener, and Okay Zed. 2013. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1057–1067.
[22] Nitin Agrawal and Ashish Vulimiri. 2017. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 647–664.
[23] Fu Bang. 2023. GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS '23)*. ACL, 212–218.
[24] Muhammad Farhan Bashir, Raja Asad Zaheer, Zohaib Mansoor Shams, and Muhammad Abdul Qadir. 2007. SCAM: Semantic Caching Architecture for Efficient Content Matching over Data Grid. In *Advances in Intelligent Web Mastering (WIC '07)*. Springer Berlin Heidelberg, 41–46.
[25] Randall G Bello, Karl Dias, Alan Downing, James J Feenan, James L Finnerty, William D Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. 1998. Materialized Views in Oracle. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., 659–664.
[26] Li Chen, Elke A Rundensteiner, and Song Wang. 2002. XCache: a semantic caching system for XML queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, 618.
[27] Dawei Cheng, Fangzhou Yang, Sheng Xiang, and Jin Liu. 2022. Financial time series forecasting with multi-modality graph neural network. *Pattern Recognition* 121, C (Jan. 2022), 108218.
[28] Boris Chidlovskii and Uwe M Borghoff. 2000. Semantic caching of Web queries. *The International Journal on Very Large Data Bases* 9, 1 (March 2000), 2–17.
[29] Boris Chidlovskii, Claudia Roncancio, and Marie Luise Schneider. 1999. Semantic cache mechanism for heterogeneous Web querying. *Computer Networks* 31, 11 (May 1999), 1347–1360.
[30] Andrew A Cook, Goksel Misirli, and Zhong Fan. 2020. Anomaly Detection for IoT Time-Series Data: A Survey. *IEEE Internet of Things Journal* 7, 7 (July 2020), 6481–6494.
[31] Shaul Dar, Michael J Franklin, Bjorn Por Jonsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., 330–341.
[32] Soumik Dasgupta, Anurag Wagh, Lalitdutt Parsai, Binay Gupta, Geet Vudata, Shally Sangal, Sohom Majumdar, Hema Rajesh, Kunal Banerjee, and Anirban Chatterjee. 2024. waLLMartCache: A Distributed, Multi-tenant and Enhanced Semantic Caching System for LLMs. In *Pattern Recognition (ICPR '24)*. Springer Nature Switzerland, 232–248.
[33] Umeshwar Dayal and Hai Yann Hwang. 1984. View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering* SE-10, 6 (Nov. 1984), 628–645.
[34] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (Aug. 2004), 5.
[35] Thibaut Germain, Samuel Gruffaz, Charles Truong, Alain Durmus, and Laurent Oudre. 2024. Shape analysis for time series. In *Advances in Neural Information Processing Systems (NeurIPS '24)*. Curran Associates, Inc., 95607–95638.
[36] Jonathan Goldstein and Per Ake Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, 331–342.
[37] Peeyush Gupta, Michael J Carey, Sharad Mehrotra, and oberto Yus. 2020. SmartBench: a benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 1807–1820.
[38] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (May 2023), 2090–2102.
[39] Yuanzhe Hao, Xiongpai Qin, Yueguo Chen, Yaru Li, Xiaoguang Sun, Yu Tao, Xiao Zhang, and Xiaoyong Du. 2021. TS-Benchmark: A Benchmark for Time Series Databases. In *2021 IEEE 37th International Conference on Data Engineering (ICDE '21)*. IEEE, 588–599.
[40] RueiJie Hsieh, Jerry Chou, and Chih Hsiang Ho. 2019. Unsupervised Online Anomaly Detection on Multivariate Sensing Time Series Data for Smart Manufacturing. In *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA '19)*. IEEE, 90–97.
[41] Soren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (Nov. 2017), 2581–2600.
[42] Chengtao Jian, Kai Yang, and Yang Jiao. 2024. Tri-Level Navigator: LLM-Empowered Tri-Level Learning for Time Series OOD Generalization. In *Advances in Neural Information Processing Systems (NeurIPS '24)*. Curran Associates, Inc., 110613–110642.
[43] Bjorn Por Jonsson, Maria Arinbjarnar, Bjarnsteinn Porsson, Michael J Franklin, and Divesh Srivastava. 2006. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology* 6, 3 (Aug. 2006), 302–331.
[44] Rui Kang and Shaoxu Song. 2024. Optimizing Time Series Queries with Versions. *Proceedings of the ACM on Management of Data* 2, 3 (May 2024), 27.
[45] Mahmoud Abo Khamis, Phokion G Kolaitis, Hung Q Ngo, and Dan Suciu. 2021. Bag Query Containment and Information Theory. *ACM Transactions on Database Systems* 46, 3 (Sept. 2021), 39.
[46] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. USENIX Association, 421–436.
[47] Abdelouahab Khelifati, Mourad Khayati, Anton Dignos, Djellel Difallah, and Philippe Cudre Mauroux. 2023. TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 3363–3376.
[48] Phokion G Kolaitis and Moshe Y Vardi. 1998. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*. ACM, 205–213.
[49] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*. USENIX Association, 1–15.
[50] Ken C K Lee, H V Leong, and Antonio Si. 1999. Semantic query caching in a mobile environment. *ACM SIGMOBILE Mobile Computing and Communications Review* 3, 2 (April 1999), 28–36.
[51] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, 447–461.
[52] Yuxuan Liang, Songyu Ke, Junbo Zhang, Xiuwen Yi, and Yu Zheng. 2018. GeoMAN: multi-level attention networks for geo-sensory time series prediction. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*. AAAI Press, 3428–3434.
[53] Jian Liu, Kefei Wang, and Feng Chen. 2021. TSCache: an efficient flash-based caching scheme for time-series data workloads. *Proceedings of the VLDB Endowment* 14, 13 (Sept. 2021), 3253–3266.
[54] Rui Liu, Jun Yuan, and Xiangdong Huang. 2024. Benchmarking Time Series Databases with IoTDB-Benchmark for IoT Scenarios. arXiv:1901.08304 [cs.DB]
[55] Qiong Luo and Jeffrey F Naughton. 2001. Form-Based Proxy Caching for Database-Backed Web Sites. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., 191–200.
[56] Jerzy Marcinkowski and Mateusz Orda. 2024. Bag Semantics Conjunctive Query Containment. Four Small Steps Towards Undecidability. *Proceedings of the ACM on Management of Data* 2, 2 (May 2024), 24.
[57] Stavros Maroulis, Vassilis Stamatopoulos, George Papastefanatos, and Manolis Terrovitis. 2024. Visualization-Aware Time Series Min-Max Caching with Error Bound Guarantees. *Proceedings of the VLDB Endowment* 17, 8 (April 2024), 2091–2103.
[58] L M D Owsley, L E Atlas, and G D Bernard. 1997. Automatic clustering of vector time-series for manufacturing machine monitoring. In *1997 IEEE International*

Conference on Acoustics, Speech, and Signal Processing (ICASSP '97). IEEE, 3393–3396 vol.4.

[59] Qun Ren and Margaret H Dunham. 1999. Using clustering for effective management of a semantic cache in mobile computing. In *Proceedings of the 1st ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDe '99)*. ACM, 94–101.

[60] Qun Ren and Margaret H Dunham. 2000. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom '00)*. ACM, 210–221.

[61] Qun Ren, M H Dunham, and V Kumar. 2003. Semantic caching and query processing. *IEEE Transactions on Knowledge and Data Engineering* 15, 1 (Jan. 2003), 192–210.

[62] Guillem Rull, Philip A Bernstein, Ivo Garcia dos Santos, Yannis Katsis, Sergey Melnik, and Ernest Teniente. 2013. Query containment in entity SQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 1169–1172.

[63] Alexander Schultheis, Lukas Malburg, Joscha Gruger, Justin Weich, Yannis Bertrand, Ralph Bergmann, and Estefania Serral Asensio. 2024. Identifying Missing Sensor Values in IoT Time Series Data: A Weight-Based Extension of Similarity Measures for Smart Manufacturing. In *Case-Based Reasoning Research and Development (ICCBR '24)*. Springer Nature Switzerland, 240–257.

[64] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. 2020. Financial time series forecasting with deep learning : A systematic literature review: 2005-2019. *Applied Soft Computing* 90 (May 2020), 106181.

[65] Veda C Storey and Robert C Goldstein. 1988. A methodology for creating user views in database design. *ACM Transactions on Database Systems* 13, 3 (Sept. 1988), 305–338.

[66] Yunxiang Su, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2024. Distance-Based Outlier Query Optimization in Apache IoTDB. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 2778–2790.

[67] Zhiqi Wang, Jin Xue, and Zili Shao. 2021. Heracles: an efficient storage model and data flushing for performance monitoring timeseries. *Proceedings of the VLDB Endowment* 14, 6 (Feb. 2021), 1080–1092.

[68] Xiaomin Xu, Sheng Huang, Yaoliang Chen, Kevin Browny, Inge Halilovicy, and Wei Lu. 2014. TSAaaS: Time Series Analytics as a Service on IoT. In *2014 IEEE International Conference on Web Services (ICWS '14)*. IEEE, 249–256.

[69] Zhenrong Xu, Pengfei Wang, Guoze Xue, Qitong Yan, Shenghao Gong, Yelan Jiang, Yuren Mao, Yunjun Gao, Shu Shen, Wei Zhang, Dan Luo, and Lu Chen. 2024. UniView: A Unified Autonomous Materialized View Management System for Various Databases. *Proceedings of the VLDB Endowment* 17, 12 (Aug. 2024), 4353–4356.

[70] Jin Xue, Zhiqi Wang, Tianyu Wang, and Zili Shao. 2022. TagTree: Global Tagging Index with Efficient Querying for Time Series Databases. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS '22)*. IEEE, 1283–1293.

[71] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., 136–145.

[72] Zhe Yang, Kun Ma, Xiaoli Zhang, Lizhen Cui, and Bo Yang. 2020. RSCVC: Row-based semantic cache with incremental versioning consistency. *Concurrency and Computation: Practice and Experience* 32, 17 (March 2020), e5672.

[73] Zhe Yang, Kun Ma, and Jialin Zhong. 2016. Toward a Semantic Cache Supporting Version-Based Consistency. In *2016 10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS '16)*. IEEE, 367–372.

[74] Xiuwen Yi, Yu Zheng, Junbo Zhang, and Tianrui Li. 2016. ST-MVL: filling missing values in geo-sensory time series data. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI '16)*. AAAI Press, 2704–2710.

[75] Geoffrey X Yu, Markos Markakis, Andreas Kipf, Per Ake Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (Sept. 2022), 99–112.

[76] Wenhan Yu and Jun Zhao. 2023. Semantic Communications, Semantic Edge Computing, and Semantic Caching with Applications to the Metaverse and 6G Mobile Networks. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS '23)*. IEEE, 983–984.

[77] Kai Zhang, Zhiqi Wang, and Zili Shao. 2023. 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)*. ACM, 10.

[78] Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. 2024. Apache TsFile: An IoT-Native Time Series File Format. *Proceedings of the VLDB Endowment* 17, 12 (Aug. 2024), 4064–4076.

[79] Bolong Zheng, Yongyong Gao, Jingyi Wan, Lingsen Yan, Long Hu, Bo Liu, Yunjun Gao, Xiaofang Zhou, and Christian S Jensen. 2023. DecLog: Decentralized Logging in Non-Volatile Memory for Time Series Database Systems. *Proceedings of the VLDB Endowment* 17, 1 (Sept. 2023), 1–14.

[80] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2015. RINSE: interactive data series exploration with ADS+. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1912–1915.