TSCache: An Efficient Flash-based Caching Scheme for Time-series Data Workloads

Jian Liu
Louisiana State University
jliu@csc.lsu.edu

Kefei Wang
Louisiana State University
kwang@csc.lsu.edu

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

ABSTRACT

Time-series databases are becoming an indispensable component in today's data centers. In order to manage the rapidly growing time-series data, we need an effective and efficient system solution to handle the huge traffic of time-series data queries. A promising solution is to deploy a high-speed, large-capacity cache system to relieve the burden on the backend time-series databases and accelerate query processing. However, time-series data is drastically different from other traditional data workloads, bringing both challenges and opportunities. In this paper, we present a flash-based cache system design for time-series data, called *TSCache*. By exploiting the unique properties of time-series data, we have developed a set of optimization schemes, such as a slab-based data management, a two-layered data indexing structure, an adaptive time-aware caching policy, and a low-cost compaction process. We have implemented a prototype based on Twitter's Fatcache. Our experimental results show that TSCache can significantly improve client query performance, effectively increasing the bandwidth by a factor of up to 6.7 and reducing the latency by up to 84.2%.

PVLDB Reference Format:
Jian Liu, Kefei Wang, and Feng Chen. TSCache: An Efficient Flash-based Caching Scheme for Time-series Data Workloads. PVLDB, 14(13): 3253 - 3266, 2021.
doi:10.14778/3484224.3484225

1 INTRODUCTION

Time-series databases are becoming an indispensable component in today's data centers [33, 34]. In recent years, we have witnessed a wide-spread use of time-series data for various applications, such as data analysis [2, 49, 67], IoT deployment [40, 72], Internet monitoring [57, 65], system alerting and diagnosis [33, 34], data mining [48, 51], visualization [59], and many others [34, 39, 41, 60].

Time-series data is a sequence of numerical data points collected over time [31, 32], such as stock prices, sensor readings, exceptional event counts, etc. Time-series data is interesting, because it provides users the ability of tracking "changes over time", which enables us to extract valuable, dynamic information that is often unavailable by solely examining each individual, static "snapshot". However, due to the nature of such data, handling a huge and quickly growing time-series dataset demands a highly efficient system design. For example, Twitter's Observability stack routinely collects 170 million metrics every minute and serves up to 200 million queries per day [11]. High-speed query processing for time-series data is highly desirable but very challenging to realize.

A simple solution is to place the entire time-series database in memory, such as Google's Monarch [34]. Such an approach, though works, would incur excessively high deployment cost, especially considering the ever-growing volume of time-series data. A more cost-efficient alternative solution is to design *a high-speed, large-capacity flash cache system* for time-series data to accelerate the queries. If the requested data can be found in the cache, a query can be completed quickly without generating extra traffic to the backend time-series database, relieving the pressure on the congested database server and improving the response time and system throughput. However, designing a highly efficient caching scheme for time-series data is non-trivial. The unique properties of time-series data bring both challenges and opportunities.

1.1 Challenges and Opportunities

Time-series data is special. Several unique properties make it drastically different from handling data in a traditional flash cache system [6, 46, 61, 70]. We must fully exploit these unique optimization opportunities and address a number of critical challenges in processing time-series data queries.

• *Property #1: Time-series data is write-once and append-only.* Time-series data is collected to record the evolution of a specific metric over time intervals. In other words, time-series data reflects the facts that happened during the past time. Such data is simply a stream of historical data points and by nature is not subject to modification later. As such, time-series data is immutable and append-only [32]. In fact, many time-series databases, such as BTrDB [41], provide a special write-once data abstraction for managing time-series data. Thus, a read-only cache would suffice. We can completely disregard data updates in cache and the related data consistence issues between cache and database, which significantly simplifies the cache design for time-series data.

• *Property #2: Range queries are dominant in time-series queries.* Time-series data is generally used for trend forecasting or behavior analysis, which are based on a collection of data points over a time range. A single data point is often uninteresting. Thus, unlike in traditional databases, range queries are typically the dominant operations in time-series databases [28, 39, 41, 63].

The dominance of range queries makes general flash caching schemes unsuitable for several reasons. First, conventional cache is mostly designed and optimized for point queries by caching data objects or blocks individually, rather than ranges of related data points as a collection. Second, conventional cache adopts an all-or-none caching method. An access to cache would be either a hit or a miss. In contrast, time-series cache must consider *partial* hit situations (some data points of a query are cached). Third, overlaps

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484225

among range query results are common, e.g., two queries have overlapped time ranges. This causes duplicate data points in cache and must be removed for an efficient use of the cache space.

• *Property #3: Time-series data queries have a unique and highly skewed access pattern.* Most time-series data applications, such as behavior analysis, trend forecasting, problem diagnosis and alerting, are particularly interested in the recently collected data [3, 39]. Facebook has reported that at least 85% of their queries are focused on data points collected in the past 26 hours [65], meaning that users' interests on time-series data are strongly correlated with data creation time, and such an interest quickly diminishes as time elapses. On the other hand, we should note that some aged data metrics could also gain interests. For instance, Etsy's Kale system periodically searches through the historical data to find anomalies with similar patterns [12, 39], which creates "hot spot" ranges over the entire time series. These two patterns could be mixed, further increasing the difficulty for making caching decisions.

1.2 Flash-based Caching for Time-series Data

In this paper, we propose a novel flash-based caching scheme, called *TSCache*, for time-series data. The goal is to provide a highly efficient caching solution to improve the query performance for time-series databases. To the best of our knowledge, this is the first work on flash-based caching for time-series data.

As a versatile distributed caching service, TSCache maintains user-supplied time-series data in high-speed, large-capacity flash storage and responds to users' requests of retrieving the stored data. Similar to Memcached [25], TSCache does not directly interact with the backend time-series database, meaning that TSCache is not an intermediate layer between the client and the database server, which allows TSCache to provide a general-purpose caching service for a variety of time-series databases simultaneously. As an abstraction of the time-series caching services, TSCache provides a simple, time range-based, key-value-like interface for users to easily store and retrieve data in the cache.

In our design, TSCache adopts a set of optimization schemes to exploit the unique properties of time-series data for caching. (1) Leveraging the write-once property, all data stored in TSCache is read-only, meaning that no changes can be made after being stored. This greatly simplifies our cache design and allows us to develop a log-like mechanism to store and manage the cache data in large-chunk slabs, which optimizes the I/O performance and reduces the management cost. (2) For optimizing range query searches, we have developed a two-layered indexing scheme to quickly filter out irrelevant data points and accelerate the time-based search in a huge pool of time-series data points in cache. (3) In order to deal with the unique access patterns of time-series workloads, we have developed an adaptive cache replacement policy to identify the most important time-series data for maximizing the utilization of the limited cache space. (4) We have also designed a low-cost compaction scheme to remove duplicate data in cache by merging overlapping data points during run time. With all these dedicated designs and optimizations, we can build a highly efficient time-series data cache.

TSCache is also highly optimized for flash storage. For example, its read-only cache design is particularly suitable for flash memory, which is known for its relatively weak write performance and lifetime issues [43, 45, 52]. The I/O operations in TSCache are parallelized, which exploits the internal parallelism of flash SSD [44, 47]. TSCache also organizes large, sequential writes by flushing in-memory slabs to flash in large chunks, which is favored by flash storage for performance and durability [38, 45].

In order to evaluate the performance of TSCache, we have implemented a prototype based on Twitter's Fatcache [10], which is an open-source flash-based key-value cache. Our implementation includes about 6,500 lines of C code for flash-based time-series cache server, 600 lines of Go code for a modified InfluxDB client, and an augmented API library based on libMemcached 1.0.18. We have evaluated TSCache with five real-world time-series datasets, including vehicle traffic, Bitcoin transactions, air pollution measurement, environmental sensoring, and taxis trajectory data. Our experimental results show that our time-series cache design can significantly improve client query performance, increasing the bandwidth by a factor of up to 6.7 and reducing the latency by up to 84.2%.

The rest of this paper is organized as follows. Section 2 gives the background. Section 3 and 4 describe the design and implementation. Section 5 presents our experimental results. Section 6 discusses the related work. The final section concludes this paper.

2 BACKGROUND

2.1 Time-series Data

Time-series data is a sequence of numerical data points collected over time [31, 32]. Each data point is collected at discrete time intervals. A timestamp is associated with each data point upon data generation. Time-series data can be classified into *regular* and *irregular* data, depending on the sampling time interval. The time interval is correlated to the data sampling rate. The higher the sampling rate is, the more data points are generated. Generally, the timestamp resolution of millisecond or second level is sufficient to satisfy most use cases. Some application scenarios demand a higher precision at nanosecond level, e.g., telemetry data [41]. As the process of data generation, collection, and distillation is highly automated, time-series data often has a high growth rate, demanding an efficient system for handling such workloads.

2.2 Time-series Databases

Time-series databases have become increasingly popular in the past few years [30]. A variety of time-series databases have been developed for different purposes [19, 26, 29, 41].

Among the time-series databases, InfluxDB ranks the top according to DB-Engines [17]. As a mature and popular time-series database, InfluxDB has been widely used for storing, analyzing, and retrieving data metrics. In the following, we use InfluxDB as a representative one to introduce the basics of time-series databases.

InfluxDB [20] is an open-source time-series database. It is fully written in Go without external dependencies. InfluxDB accepts SQL-like queries, called *InfluxQL*, via HTTP or JSON over UDP. It supports simple data aggregations but complex join operations are not supported. Its storage engine adopts a data structure called TSM-Tree, which is similar to LSM-Tree [64], to optimize insert operations. InfluxDB does not support delete, but it removes expired data according to the specified retention policy.

In InfluxDB, the time-series data is stored in *shards*. A shard usually covers a specific time range of data points. Each shard is

Table 1: An illustration of measurement in InfluxDB [21].

Time	Btrfly	Hnybees	Location	Scientist
15-08-18T 00:00:00Z	12	23	1	langstroth
15-08-18T 00:00:00Z	1	30	1	perpetua
15-08-18T 00:06:00Z	3	28	1	perpetua
15-08-18T 00:06:00Z	1	30	1	langstroth

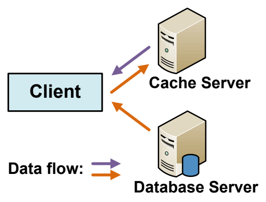

Figure 1: An illustration of data flow in TSCache.

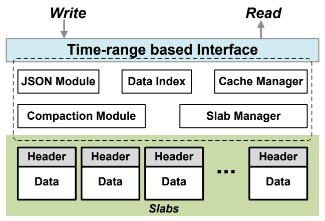

Figure 2: An illustration of TSCache structure.

an independent data management unit, which has its individual cache, Write Ahead Log (WAL), and TSM files. To guarantee data reliability, the incoming data is first written into WAL and then stored in TSM files on disk. Each TSM file is a read-only file storing compressed time-series data in a columnar format. The structure of a TSM file is similar to SSTable in LevelDB [24] and includes four sections, namely header, blocks, index, and footer. Multiple shards with disjoint time ranges can be further organized in a *shard group* and share the same retention policy. Expired shards are removed from the database to release resources.

InfluxDB is schemaless. Users can add *measurement*, *tag set*, and *field set* to database at any time according to their needs. InfluxDB manages time-series data in *measurement*, which is akin to "table" in a traditional SQL database. Each data point is associated with a timestamp upon data generation. Different data points can have the same timestamp. Each data point has one or multiple *field keys* and one or multiple *tag keys*. All data points are sorted in the time order when stored in the measurement. Here we use an example shown in Table 1 for explanation.

Table 1 illustrates an example that measures the census of butterflies and honeybees by two scientists in different locations. This measurement has two field keys, Butterflies and Honeybees. The data in the two columns represents the actual *field values*. Note that field keys are strings, while field values can be integers, strings, floats, etc. A unique combination of field values for different field keys is also called a *field set*. In this example, we have four time-series data points in three different field sets, of which the field set (butterflies=1, honeybees=30) has two data points. It is worth noting that field set cannot be indexed. Using an improper field value as the filter condition may incur a long search time.

Each data point can also be associated with one or multiple *tag keys*. Two tag keys in the measurement are location and scientist, each of which is associated with their *tag values* ("1" for location, and "langstroth" and "perpetua" for scientist). A *tag set* is a unique combination of tag values for different tag keys. In this measurement, we have two tag sets in total, (location="1", scientist="langstroth") and (location="1", scientist="perpetua"). Both tag key and tag value can only be strings, being used as metadata to describe the field value. Different from field set, tag set can be indexed, which improves the query efficiency with a proper setting.

Table 2: An illustration of series in InfluxDB [21].

Series	Measurement	Tag Set [Loc., Sci.]	Field Key
series 1	census	[1, langstroth]	butterflies
series 2	census	[1, perpetua]	butterflies
series 3	census	[1, langstroth]	honeybees
series 4	census	[1, perpetua]	honeybees

A *series* refers to a logical group of data points which share the same measurement name, tag set, and field key. In Table 2, we have four series in the measurement in total. In other words, the series defines the structure of a time-series dataset. Understanding the series concept is useful for practitioners to achieve high efficiency of operations on time-series database.

3 DESIGN

In this paper, we present a highly efficient time-series cache system design, called *TSCache*, to accelerate client queries to time-series databases. The goal of TSCache is to provide independent, network-based caching services for time-series database clients.

As shown in Figure 1, a typical workflow with TSCache is similar to Memcached—Upon a time-series query, the client first queries the TSCache server. If the data is found in cache, the cache server retrieves the data and returns to the client; otherwise, the client receives NOT_FOUND and needs to directly query the backend database, and then it sends the received time-series data to the cache server for caching and serving future queries. Upon insertion, data points are directly inserted into the backend database without involving TSCache, meaning that TSCache is not in the critical path.

Such a design brings two important advantages for time-series workloads. First, it detaches the cache from any specific database, allowing us to provide a general-purpose, shareable caching service for various types of time-series databases. Second, it is similar to popular key-value caching services, such as Memcached [25] and Redis [27], which many developers and practitioners are already familiar with, enabling an easy and smooth adoption in practice.

3.1 Architecture Overview

As illustrated in Figure 2, TSCache has five major components: (1) *Time-range based interface*: A general, simplified interface is provided for clients to store and retrieve time-series data, which is typically a collection of data points returned from a time-series database for a query. (2) *Slab manager*: TSCache adopts a slab-based management to accommodate incoming data in large, write-once-read-many chunks as a sequence of continuous data points in the time order. (3) *Data index*: A two-layered data indexing structure is used to quickly filter out irrelevant data points and accelerate time-range based searches. (4) *Cache manager*: An adaptive cache replacement scheme is designed to identify the most valuable data for caching, which is optimized for the unique access patterns in time-series workloads. (5) *Compaction module*: A low-cost compaction process runs in the background to remove duplicate data points for optimizing cache space utilization.

3.2 Time-range based Interface

Traditional cache systems are designed for caching data objects or chunks individually. Each data item in cache can be located by an identifier, such as a key, an object ID, or a block number, etc.

Time-series caching is different. A client typically caches the query result for a specified time range, which is a logical dataset that contains a sequence of data points. However, we cannot directly use query as a unit for caching and management, because the time range of interest often changes. For example, a client may need to zoom in/out and examine a sub-range of the time-series data.

To address the above issue, the data in TSCache is identified by two basic properties, *query* and *time*. The former defines a specific query condition, and the latter defines a contiguous time range. In this way, we can track for a specific query, which time range of data points is available in our cache server.

Interface design. Inspired by key-value caches, TSCache provides a *Time-range based, Key-value-like Interface* for clients to store and retrieve data. Specifically, we use the data query statement excluding the time range part to calculate a 160-bit SHA-1 hash value [1] as a *key* to represent the query. We provide an API function in libMemcached library as a tool to help users quickly generate a hash key for a given query statement. Then, the key is used as a handle to interact with the cache server. Due to the SHA-1 hashing, the query can be uniquely identified with the key.

This approach brings several benefits. First, TSCache does not need to understand the specifics of a query, but only needs to differentiate queries using the keys, which avoids parsing complex data query statements in the cache system. Second, it allows TSCache to be seamlessly integrated with a variety of time-series databases, which may use totally different query languages. Third, this interface mimics the widely used key-value interface, making it easier to be adopted in practice by users.

API functions. TSCache provides two basic operations for users to set (store) and get (retrieve) data in the cache, as follows.

`Set(key, value, time_start, time_end)` stores the client-supplied data to the cache server. The key is a 160-bit digest calculated using the SHA-1 cryptographic hash function based on the data query statement. The value is a JavaScript Object Notation (JSON) [23] file with a collection of data points encoded, which is shareable across different platforms. The time range of the data points is specified by `time_start` and `time_end`. The key and the time range pair together uniquely identify the value. The client is responsible for guaranteeing that the supplied data points are the complete result of a database query for a specified time range.

`Get(key, time_start, time_end)` retrieves data points from the cache server according to the provided key and time range. The key represents the query, and the pair of `time_start` and `time_end` specifies the requested time range. To retrieve data points from the cache server, it needs to meet two conditions: The key should be matched, and based on that, the requested time range should be covered by the cached data. The data points returned to the client are encoded as a JSON file to guarantee its integrity.

3.3 Handling Partial Hits

Unlike in traditional cache, where a requested data object is either found in cache (a hit) or not (a miss), a special situation, *partial hit*, may happen in time-series cache. For example, the requested time range cannot be completely covered by the data in cache.

We have two possible options to handle partial hit situations: (1) The cache returns the partially found data to the client, which then submits another query to the database to retrieve the missing part, or (2) the cache notifies the client `INCOMPLETE`, and the client then submits the full query to the database to load the complete data.

Handling partial hits. To understand the performance impact of the above two choices, we have designed an experiment to compare their I/O efficiencies. In order to simulate the partial hit scenario, we split a request into two operations, first reading from the cache and then from the backend database in a sequence, and calculate the sum of the incurred latencies for each request. More details about the system setup are available in Section 5.

Figure 3 shows the average latency of reading data from the cache server and the database server with different request sizes. For requests smaller than 512 KB, the time spent on the database server dominates the query latency, meaning that splitting a relatively small request does not bring much performance benefits. However, for large requests (over 512 KB), retrieving data even partially from the cache server is helpful. For 4-MB requests, reading half from cache can reduce the average latency by 36.2%, compared to reading it all from the database. This result indicates that the benefit of loading partial data from cache depends on the request size.

TSCache adopts an adaptive approach to handle partial hits. (1) If the requested data size is small (less than 512 KB in our prototype), we treat partial hit simply as a miss and quickly respond to the client without further searching the cache, and the client needs to submit a full request to the database server. (2) For large requests, we return the partial data found in the cache server to the client, and the client needs to create another partial request to load the remaining data from the database server.

It is worth noting here that partial hit happens whenever two queries have partially overlapping data, disregarding their creation time (new or old). TSCache selectively returns the cached data only when it is cost beneficial to retrieve the partially hit data, which avoids unnecessary overhead as discussed above.

Request size estimation. The adaptive handling of partial hits demands an estimation of the data amount for a request. If the partial data is too small, we desire to quickly respond to the client `INCOMPLETE`. In reality, however, it is difficult for the cache server to make an accurate estimation, because for a given request, both the number of involved data points and the size of each data point are unknown. We could simply scan the cache to look up the involved data points, but this approach is clearly very time-consuming and foils the effort of quickly responding to the client.

TSCache estimates the data size as follows. Upon a data write to the cache server, we know the time range T and the data amount V, based on which we can calculate an estimated *data density*, $D = V/T$. The density information is associated with the time range and stored in the key index as part of the metadata (see Section 3.5). Upon a read request for a time range t, the data amount is estimated as $t \times D$. In this way, we can quickly estimate how much data would be involved in the query and decide whether we should handle a partial hit as a miss by only accessing the key index.

Fragmented requests. Allowing partial hits may lead to a more complicated situation, which is that a request could split into many small, disjoint ranges. A simple example is, assuming the requested time range is [0, 100] and the cache covers the time range of [25, 75], then the client has to generate requests to the database server for two time ranges of [0, 25] and [75, 100]. The real-world scenarios could be more complex, creating several critical challenges.

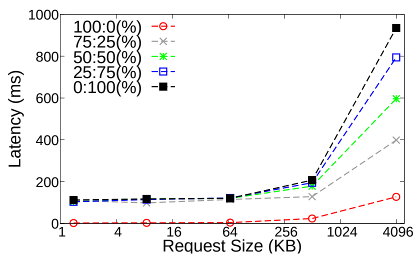

Figure 3: Average latency of split requests. M:N means the ratio of data split to the cache and the database.

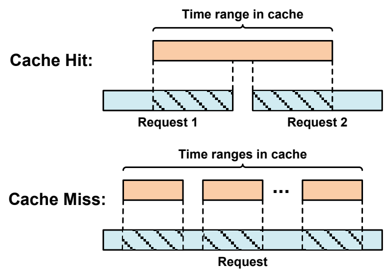

Figure 4: Partial hits. Blue and orange boxes represent the requested and cached time ranges, respectively.

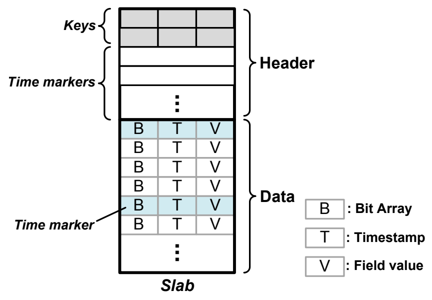

Figure 5: The structure of a slab.

First, the client must be prepared to handle a variety of possible cases, analyze received data, generate multiple requests, merge received data, etc. It puts an excessive burden on programmers, discouraging them to adopt this caching service. Second, the cache server design becomes too complex. The cache has to search all possible disjoint time ranges. Third, a more complex protocol is needed for client-cache communications to represent and deliver these multi-range data. Finally, and most importantly, the performance gain of handling many small requests is in question, especially considering the involved overhead and additional I/Os.

We tackle this problem by using a simple solution: The cache returns partial data only if the head or the tail part of the requested time range is covered in cache (see Figure 4). In all other cases, we handle it as a miss. It guarantees that one query would split into at most two parts, one to the cache and the other to the database.

This design greatly simplifies the problem. First, the client becomes simple. It only needs to concatenate the two parts together. Second, the cache server is also simplified and only needs to handle one continuous time range query. Third, the protocol is also simplified. We only need to specify the time range in the returned JSON file. Finally, the size estimation is simpler and becomes feasible.

3.4 Slab Management

Time-series data is write-once and append-only. We take advantage of this property and use a log-like mechanism to manage data in large chunks. In TSCache, the basic unit for data management is a *slab*, which stores a sequence of time-series data points in their time order. Thus each slab represents a time range of data.

The slab-based management brings several advantages. First, it creates sequential, large writes, which is beneficial in terms of write performance and lifetime for flash devices [38, 45, 61]. Second, for range queries, organizing consecutive data points in a large slab improves I/O efficiency. Third, managing data in large units can effectively reduce the management overhead.

Slab structure. As shown in Figure 5, each slab is an individual, self-contained unit, which stores a sequence of data points and the associated metadata for a continuous time range. A slab consists of two parts, *header* and *data*. The header part contains the metadata that describes the data part and facilitates a quick search for a query; the data part stores a list of data points sorted in the time order.

To understand the rationale of our slab-structure design, we first explain a critical challenge that we must address. In time-series cache, a cached data point could be associated with multiple queries, each represented by a query key. Simply mapping a key to an individual set of data points would result in severe data redundancy in cache, since the query results may overlap. Alternatively, we could map multiple keys to each shared data point. However, due to the huge number of data points, building a many-key-to-many-data-points mapping structure is unrealistic. Simply recording each query's start and stop positions is also infeasible, since data points of different queries may be mixed due to time range overlaps.

We use a *Label-based Reverse Mapping* to solve the problem. We associate each data point with a set of flag bits (128 bits), called *bit array*. Each bit corresponds to a query key in the slab. A set bit means the data point is associated with the query. The bit array essentially labels the keys of a data point in a space-efficient way. To complete the reverse data-point-to-keys mapping, we need another structure in the slab header as described below.

Key array. The first part of the slab header is a reserved area for a list of 128 SHA-1 keys, called *key array*. The key array is a collection of unique query keys, each of which has at least one data point appearing in the slab. Each 160-bit key corresponds to one bit in the above-mentioned bit array.

The key array and the bit array together map each data point back to the associated keys, bringing two benefits. First, it allows us to attach a small bit array (128 bits) to label each data point, which is simple and space efficient. Second, organizing all resident keys in the header allows us to load a small header to quickly filter out a query that does not have data in the slab, which avoids reading the entire slab and improves the speed.

The number of keys allowed in a slab can be extended. In general, 128 keys are sufficient for a slab in most cases. If a slab needs to accommodate more than 128 keys, we may copy the slab to a temporary buffer in memory, double the key array and bit array size to 256 entries, then copy the data points back, and update the related header information. Due to the extra space needed for the additional keys, we may need to create an extra slab to contain all the data points in the original slab.

Time markers. The second part of the slab header maintains a special structure to help quickly locate the position of the target data in the slab. A slab stores a large number of data points in the time order. However, a query may only need to access a sub-range of them. Since the sizes of data points vary, it is difficult to estimate the position of the first data point in the target time range. Scanning the data points would incur read amplification problem.

To accelerate this process, for each 4 KB block of the data part, we store a *time marker* in the slab header to track the timestamp and the in-slab offset of the first data point of this block. In our

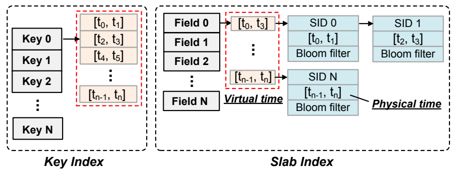

Figure 6: Two-layered data indexing.

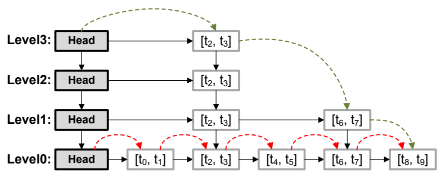

Figure 7: Time-range based skip list.

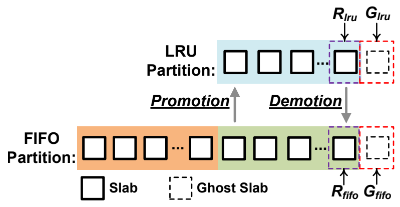

Figure 8: Cache replacement policy.

current setting, we reserve space for storing 256 time markers. For a query that specifies a time range, we simply read the list of time markers, and locate the one closest to the start time of the requested time range and read data from there. It allows us to avoid reading the unwanted data, significantly reducing read amplification.

Data part. The rest of the slab stores data points. We adopt a column-based style. Each slab only stores data points of a specific field (a column). This is for two reasons. First, we can simultaneously read multiple slabs containing different fields, which creates parallel I/Os on flash and accelerates the request. Second, it enables us to address a unique issue in time-series caching, which is how to combine incomplete data points together. A query to time-series database may not always request all fields. In the cache, if we store data points in the row-based manner, many "holes" would appear in a row, which would cause significant space waste.

3.5 Two-layered Data Indexing

An efficient data indexing structure is crucial to the query performance. In TSCache, we use a two-layered in-memory data indexing structure to fulfill this requirement. As shown in Figure 6, two separate indexing structures, *key index* and *slab index*, are independent from each other and designed for different purposes.

Key index. The key index structure keeps track of the queries that have data cached in the system. Its main purpose is to facilitate us to quickly filter out cache misses, either in cases when the query key is not found in the cache, or the key is found but the requested time range is not found.

The key index structure is organized as a hash table with skip lists. The first level is a traditional hash table indexed by the query keys. Each key is associated with a sequence of individual time ranges, which is organized in a skip list structure [66]. When adding a new time range into the cache, any overlapping time ranges would be merged, ensuring that the key's time ranges are non-overlapping.

For a given key, as shown in Figure 7, its time ranges are organized as a 4-level skip list. Each item represents an individual time range [`time_start`, `time_end`]. Each level is a singly linked list. The bottom level (Level 0) is a complete list of all the time range items sorted in the ascending time order. Each of the upper levels (Level 1 to Level 3) provides a list of reference points to selected items. The higher the level is, the shorter the list is. The structure of skip list allows us to quickly skip unrelated time ranges and locate the target time range, which accelerates the search process in a long list. We find that a skip list of 4 levels is sufficient for our workloads. More details about skip list can be found in prior work [66].

Slab index. The slab index keeps track of all the slabs in the cache. For each slab, we maintain a 28-byte *slab node* in memory to manage its metadata, including a 4-byte slab ID (SID), a 8-byte Bloom filter, and two 8-byte timestamps, which represent the physical time range of the data points in the slab.

Similar to column store databases, TSCache manages *fields* (akin to "column" in a traditional database) in separate slabs. Each slab contains a sequence of data points for a specific field. Thus, the first level of the slab index is organized as a hash table indexed by the field number. As a typical time-series measurement has only 4 to 10 fields, a 16-bucket hash table is sufficient for most cases.

A field is associated with a list of non-overlapping *virtual time ranges*, each of which corresponds to a set of slabs whose data points together cover a continuous range of time. We connect the slab nodes belonging to a virtual time range using a singly linked list in the ascending time order. Note that the slabs in a list may have overlapping *physical time ranges*. A compaction procedure periodically runs to remove the duplicate data (see Section 3.7).

In order to quickly search for the target key and time range in the list of slabs, we also leverage the Bloom filter and the physical time range information in slab nodes. (1) In a slab node, the Bloom filter tracks the query keys whose data points are stored in the slab. If the Bloom filter indicates that the target key is not in the slab, we simply skip it and examine the next slab; otherwise, (2) we further compare the target time range with the slab's physical time range, which records the first and last timestamps in the slab. If the requested time range is completely or partially in the slab's physical time range, we then read the slab from the cache; otherwise, we skip the slab and examine the next one in the list. In this way, we can quickly filter out irrelevant slabs without incurring unnecessary I/Os (for reading slab headers), improving the cache lookup speed.

Handling a query. Upon a cache query with a key and a specified time range, it works as follows. We first search the key index structure to determine if the key is found in cache. If true, we further search the key index to determine if the target time range is cached. If the time range is partially hit, we run the request size estimator (see Section 3.3) to decide if we return partial data to the client. In the case of a hit or a partial hit, we further turn to the slab index structure. For each requested field, we first locate the involved virtual time ranges, and then locate the involved slabs. Then we read the slab data and finally return it to the client.

3.6 Time-aware Caching Policy

Time-series data workloads exhibit a unique, mixed access pattern: The more recently created data typically gain more popularity than old data, while some relatively aged data also exhibits high popularity as "hot spot" time ranges. We introduce a *Time-aware Caching Policy* for managing time-series data in TSCache.

Cache replacement policy. As shown in Figure 8, we logically divide the cache into two partitions, which are managed separately using different caching policies.[1] The upper-level *LRU partition*

[1]Note that this division is logical. Moving a slab from one partition to the other only needs metadata changes and does not involve physical data movement.

maintains a list of slabs that contain relatively old but "hot" time range data. This partition uses the traditional LRU replacement algorithm. The lower-level *FIFO partition* maintains a list of slabs that are arranged in the time order and managed using a simple FIFO replacement algorithm. The partition sizes are adaptively auto-tuned according to workloads during run time.

Our caching policy works as follows. When a slab is added into the cache, it is first inserted into the lower-level FIFO partition. Since most time-series workloads are more interested in the recent data, we adopt the First-in-first-out (FIFO) replacement policy, meaning that the oldest slab would be evicted first. We divide the FIFO list into two equal-size parts (old and new). Upon an re-access to a slab, if it is in the first half (newer) of the FIFO list, the slab remains in its current position in the FIFO list with no action; otherwise, it is promoted into the upper-level LRU partition. In the meantime, the least recently used (LRU) slab in the LRU partition is demoted into the FIFO partition and inserted into the list according to its time. For eviction, we always choose the oldest slab at the tail of the FIFO list as the victim slab for eviction.

The design rationale is as follows. We use the FIFO partition to manage most slabs in their original time order, which ensures that the more recent slabs to stay in the cache longer and the old data would be evicted first. However, if a relatively old slab is re-accessed, it means that this slab is likely to cover a hot time range. So we promote it to the LRU region and give it a second chance to stay in cache for longer time. If a slab in the LRU list cools down, it would be demoted to the FIFO list and eventually evicted.

Auto-tuning partition sizes. Partition sizes can affect performance. The larger the LRU partition is, the cache replacement is more like the traditional LRU. We set the initial sizes of the LRU and FIFO partitions at the ratio of 1:2, similar to the inactive and active lists in Linux kernel [54]. We also design an adaptive, auto-tuning method to adjust the partition sizes during run time.

We attach a *ghost slab* to the end of the LRU partition and the FIFO partition, called G_{lru} and G_{fifo}, respectively. The ghost slab does not contain actual data but only the metadata of the last evicted slab. We also keep track of the number of hits to the least recently used slab resident in the LRU partition and the oldest slab resident in the FIFO partition, called R_{lru} and R_{fifo}, respectively. Each time when a slab is evicted, we also make the size-tuning decision based on the observed hits on the four above-said slabs as follows.

Let us use H to denote the number of hits in a specified slab. There are three possible situations. (1) If $H(G_{lru}) - H(R_{fifo})$ is greater than $H(G_{fifo}) - H(R_{lru})$, it indicates that taking a slab from the FIFO partition and giving it to the LRU partition can bring benefits in increasing the number of slab hits. Thus we enlarge the LRU partition size by one slab and reduce the FIFO partition size by one slab. (2) If $H(G_{lru}) - H(R_{fifo})$ is lower than $H(G_{fifo}) - H(R_{lru})$, we take a slab from the LRU partition to the FIFO partition. (3) Otherwise, we keep the current partition size unchanged.

3.7 Compaction

To quickly handle the ingress flow, TSCache always allocates free slabs to accommodate incoming data points for caching. This design creates large, sequential I/Os and significantly improves write performance, but it also incurs several issues.

First, since different queries' results may overlap, duplicate data points may exist in slabs. Such redundancy must be removed to save cache space. Second, the space of a slab may not be completely used (the tail part). Third, having more slabs also means a longer list of slabs in the slab index, which slows down the search process.

In order to address these issues, TSCache periodically runs a *Compaction* procedure in the background to remove duplicate data points. It works as follows. Each virtual time range is associated with a slab node list. If the number of slabs on the list exceeds a predefined threshold (e.g., 16 slabs), a compaction procedure is initiated to merge slabs and remove duplicate data points. We follow the reverse chronological order (the new slabs first) to read and merge data points. This process repeats until the entire list of slabs is scanned and merged. If multiple virtual time ranges need to be compacted, we compact each of them, also starting from the most recent one and following the same procedure.

It is worth noting an optimization here. Due to the unique properties of time-series data, relatively old time-series data is more likely to be removed out of cache soon. Compacting such to-be-removed data is clearly an unnecessary waste of time. Thus, we always start compaction in the reverse chronological order (the new slabs first). For each virtual time range, we read the slabs and merge the data points into a new list of slabs. In this process, since a data point could be associated with more than one key, the related bit array needs to be updated. It is also worth noting that depending on the number of keys in a slab, we can flexibly expand the header part of a slab as described in Section 3.4.

4 IMPLEMENTATION

In order to evaluate the performance of TSCache, we have implemented a time-series cache prototype based on Twitter's Fatcache [10]. Our prototype adds about 6,500 lines of C code. In the prototype, the time-series data is managed in the restructured slabs to handle time range queries. We have also modified the original hash table by using the proposed time-range based skip list for data indexing. Fatcache originally adopts a synchronous FIFO-based replacement policy. Our prototype implements an asynchronous, two-level replacement algorithm customized for time-series data.

For communications between the client and the cache server, our TSCache prototype encodes all data points of a query in a JSON file by using cJSON library [5]. One change we made to the library is that the original `integer` variables in cJSON structure is replaced with `long` integer types to accommodate long timestamps. Since JSON is widely supported in various language libraries on different platforms, using JSON files for client-cache communications ensures cross-system compatibility and allows our TSCache server to support a diverse set of time-series databases.

We have also implemented a client manager, adding about 600 lines of Go code based on the InfluxDB client [22], to access the cache server and the backend InfluxDB database server. To support time range query and also maintain high scalability, we have implemented our time-series cache API interface based on libMemcached 1.0.18 [8]. The added API support allows a client to deliver the time range parameters to the cache server. We use the `cgo` method in our Go client to call the C functions provided in libMemcached for communications with the cache server.

Table 3: The characteristics of time-series datasets.

Name	Traffic	BTC	Pollution	Sensor	Trajectory
Sampl. interval (Sec)	300	24	300	30	300
Num. of fields	9	7	8	8	5
Duration	Jan. 2006~Nov. 2014	Jan. 1980~Sep. 2020	Jan. 2010~Oct. 2014	Jan. 1996~Apr. 2004	Oct. 2007~Feb. 2008
Num. of data points	288,467,566	141,619,759	215,201,210	257,654,957	389,319,566
Total size (GB)	21.0	13.5	20.7	20.2	20.9

5 EVALUATION

5.1 Experimental Setup

We have conducted a set of experiments to evaluate our caching scheme. Our test platform includes a client, which generates time-series data requests, a TSCache server, which serves data cache requests, and an InfluxDB server with default settings, which serves as the backend time-series database.

System setup. Our experiments are performed on three Lenovo TS440 ThinkServers with 64-bit Ubuntu 18.04 LTS systems. Each server is configured with a 4-core Intel Xeon E3-1266 3.3-GHz processor, 12 GB memory, and a 7,200 RPM 1-TB Western Digital hard drive. The cache server is equipped with a 400-GB Intel 750 PCIe flash SSD. The SSD device is directly controlled by the cache manager with no file system. All read and write operations are executed in `direct_io` mode to bypass the OS page cache. In the database server, a separate 7,200 RPM 2-TB Seagate hard drive is used to host an InfluxDB 1.1.1 database as the backend to avoid the interference with the system drive. The three servers are connected in a 10-Gbps Ethernet network.

Datasets. Five real-world time-series datasets are used for synthesizing realistic time-series data in our experiments. *Traffic* [14] is a collection of vehicle traffic data in the city of Aarhus, Denmark over a period of six months. *BTC* [16] records Bitcoin transaction information between a buyer and a seller, covering a duration of over 1 year. *Pollution* [13] is a collection of air pollution measurements that are generated based on the Air Pollution Index [9]. *Sensor* [4] is collected from 54 sensors deployed at the Intel Berkeley Research Lab. *Trajectory* [75, 76] is a sample of T-Drive trajectory dataset, which contains one-week trajectories of 10,357 taxis. The original datasets are relatively small. In order to fully exercise our system, we synthesize larger datasets by repeating each original dataset multiple times to cover a longer duration, and the original sampling rate, fields, etc. remain unchanged. More details are in Table 3.

Workloads. We use time-series benchmark YCSB-TS [7] to generate queries with different time ranges for our experiments. Each query has a time range [`time_start`, `time_end`] following Latest distribution. The time range length of each query follows Zipfian distribution between `maxscanlength` and `minscanlength` as defined in configuration. All queries excluding the time range part are used as keys for caching, following Zipfian distribution.

We configure a *Full* workload, the queries of which have time range varying from one minute to one week, representing a typical use case in practice. The average request size is 15.11 KB, 13.68 KB, 15.15 KB, 10.12 KB, and 24.77 KB for Traffic, BTC, Pollution, Sensor, and Trajectory, respectively. For each dataset, we generate 100,000 queries with variable time ranges following Zipfian distribution as defined in YCSB-TS. We use this workload for our experiments in overall comparison and component studies. For comparison with Fatcache, we also configure three extra workloads to request time-series data in different time ranges (see Section 5.2). In our experiments, we find that an excessively large number of clients can overload the server in the database-only case. We use 64 client threads to run the workloads, which allows us to fully exercise the system without over-saturating it. The cache size, unless otherwise specified, is set to 10% of the workload's dataset size.

5.2 Overall Comparison

In this section, we show the overall performance of TSCache for speeding up queries using a set of time-series data workloads. We measure the performance using three key metrics, *hit ratio*, *bandwidth*, and *latency*. To ensure each experiment begins at the same state, we first create a database and load data points into the database, and in each run, we start with the same loaded database.

We compare TSCache with three other system solutions. (1) *Single database server*. Our baseline case is a non-caching, database-only system which runs a single InfluxDB server, denoted as *Single-DB*. (2) *Two database servers*. TSCache uses an additional cache server. For fair comparison, we also configure a two-server database-only system, in which two identical database servers are deployed to share the workload of incoming queries in a round-robin manner, denoted as *Dual-DB*. For the two database-only systems, we also configure an SSD-based setup, denoted as *Single-DB-SSD* and *Dual-DB-SSD*, respectively, for studying the effect of storage devices. (3) *General-purpose cache*. To compare with a general-purpose caching solution, we have implemented a simple time-series cache based on Twitter's Fatcache [10], denoted as *Fatcache*. The time-series data stream is divided into chunks using fixed-size time units, similar to prior work [71]. Each chunk is stored in cache as an individual caching unit indexed by its query and time range.

Overall comparison. Figure 9 shows the overall performance of the four system solutions. Compared to the two database-only systems, TSCache achieves the best overall performance across all the five workloads. By caching only 10% of the dataset, TSCache improves the bandwidth by a factor of 2.7–6.7 and reduces the average latency by 63.6%–84.2% than Single-DB. If comparing with Dual-DB, the two-server setup, TSCache increases the bandwidth by a factor of 1.7–3.4 and reduces the latency by 42.3%–68.6%.

TSCache is also much more efficient than simply adopting a general-purpose cache. In this test, we use a 4-hour time unit to chunk time-series data in Fatcache. In Figure 9(a), we can see that TSCache achieves a significantly higher hit ratio than Fatcache. TSCache achieves a hit ratio of 88%–92.1% under the five workloads, which is an increase of 30.8–59.9 percentage points (p.p.) than Fatcache. As a result, TSCache improves the bandwidth by a factor of 1.4–3 and reduces the average latency by 22%–66.9%.

TSCache vs. Fatcache. To understand the performance gains in more details, we further compare TSCache with Fatcache using

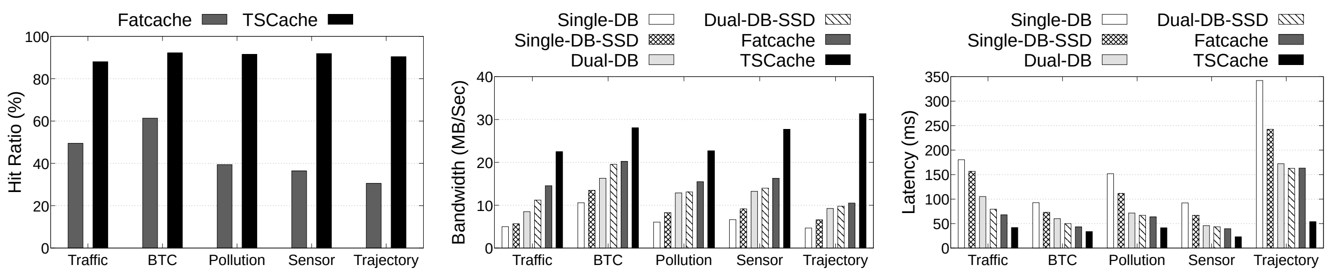

Figure 9: Hit ratio, bandwidth, and latency comparison between different system solutions under five data workloads.

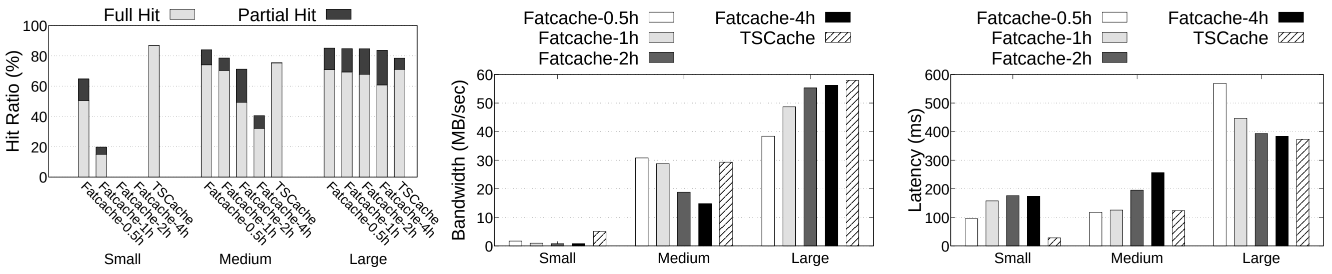

Figure 10: Hit ratio, bandwidth, and latency comparison between Fatcache and TSCache under Trajectory data workload.

three special workloads on Trajectory dataset, *Small*, *Medium*, and *Large*, which request time-series data in a time range of 1 minute to 1 hour, 1 hour to 1 day, and 1 day to 1 week, respectively. The average request size is 2.28 KB, 58.82 KB, and 348.91 KB for Small, Medium, and Large workloads, respectively. We also configure four time unit sizes, from 0.5 hour to 4 hours, for chunking time-series data in Fatcache. In general, the more fine-grained chunking is, the higher hit ratio but also higher overhead would be.

As shown in Figure 10, under Small workload with Trajectory dataset, TSCache achieves a hit ratio 22.2–86.9 p.p. higher than Fatcache, improving the bandwidth by a factor of 3–6.5 and reducing the latency by 70.6%–84.1%. For Fatcache, small requests cannot benefit from cache due to the fixed-size chunking. For chunking in 2-hour units (Fatcache-2h) and 4-hour units (Fatcache-4h), the hit ratios even drop to zero. Under Medium workload, although the hit ratio of TSCache is 8.5 p.p lower than the best Fatcache configuration (Fatcache-0.5h), it achieves comparable bandwidth and latency. Fatcache gains more partial hits, but due to the request splitting, it does not translate into observable performance benefits. This effect is more evident with Large workload. Although the more partial hits in Fatcache bring 5.2–6.6 p.p. higher hit ratios than TSCache, the bandwidth of TSCache is 3%–50.8% higher than Fatcache and the latency is 2.9%–34.6% lower. It is mainly because Fatcache has to split a large time range query into many small ones, which incurs severe I/O amplification. In fact, the number of requests is amplified by 21.5 times for Fatcache-0.5h, which overloads the servers. This result clearly shows that simply using general-purpose caching is inefficient for time-series data.

HDD vs. SSD. We have also configured an SSD-based setup for the two database-only systems and compare with TSCache. As shown in Figure 9, using SSD provides limited performance benefits for the two database-only systems, improving the bandwidth by up to 41.1%. TSCache outperforms them both by a factor of 1.4–4.8. It is mainly because InfluxDB is designed for large-capacity storage and not optimized for SSD. It also applies aggressive data compression for space saving, which diminishes the impact of storage in overall performance. TSCache, in contrast, is tailored to SSD's properties, such as creating parallel I/Os, large and sequential writes, etc., and it also caches data in their original, uncompressed format, which leads to better performance. Further considering the cost issues, holding the entire time-series dataset in SSD would be an expensive and inefficient solution in practice.

Performance vs. cache size. We also measure the performance effect of cache sizes in Figure 11. We configure the cache size from 4% to 10% of the workload's dataset size. As the cache size increases, the bandwidth and hit ratio also increase, and the latency decreases. In particular, as we enlarge the cache size, TSCache sees an increase of hit ratio by 23.7–48 p.p., which in turn leads to the bandwidth increase by a factor of 1.6–2.4 and the latency decrease by 34.2%–57.6%, which clearly shows the efficacy of the cache system.

5.3 Scalability

Scalability is important for handling an increasing amount of time-series data queries. As shown in Figure 12, as we increase the number of clients from 1 to 64, the overall bandwidth increases by a factor of 6.9–9.1 under the five workloads. Since we can run multiple queries in parallel, it allows us to fully utilize the system resources, such as the rich internal parallelism of flash SSD [44, 47], and thus reduce the total execution time to finish the same amount of query jobs. On the other hand, the average latency also increases by a factor of 7–9.4 due to the increased delay among queries.

5.4 Slab Management

TSCache designs a unique slab structure for time-series data management. In this section, we evaluate two key components, time markers and parallelized I/O operations.

5.4.1 The Effect of Time Markers. In order to accelerate the in-slab data lookup, TSCache includes a special structure in slab to mark the timestamp of the first data point appearing in each block of its data part. The purpose is to skip irrelevant blocks and quickly locate the target time range without reading the entire slab. This experiment compares TSCache performance with and without time

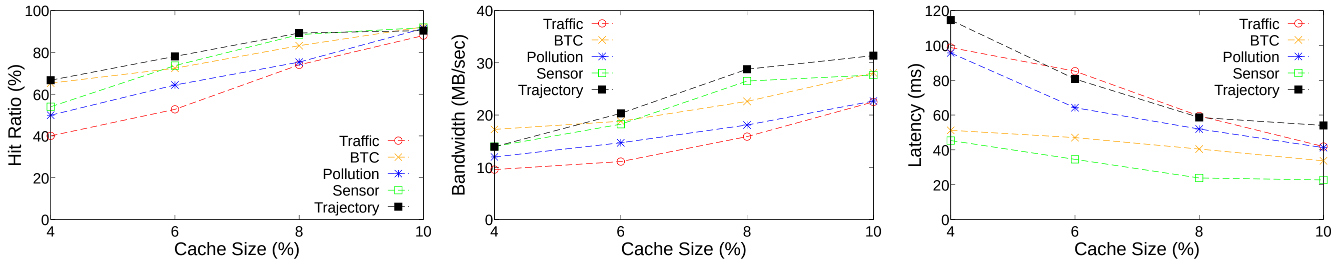

Figure 11: Hit ratio, bandwidth, and latency of TSCache with different cache sizes.

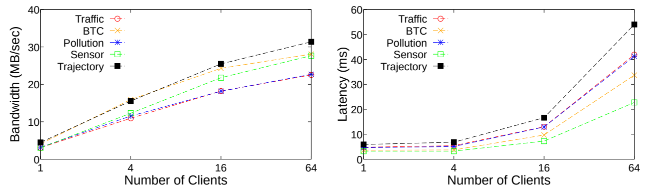

Figure 12: Performance effect of the number of clients.

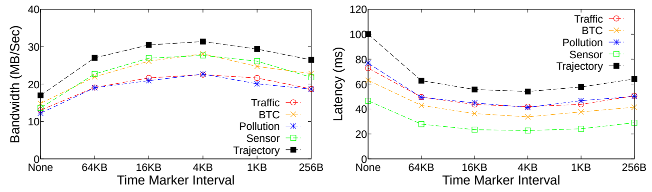

Figure 13: Performance effect of time marker interval.

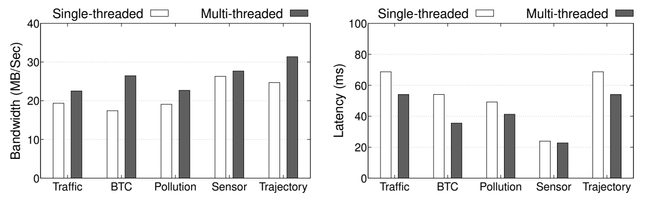

Figure 14: Performance of TSCache with multi-threading.

markers. We also study the impact of time marker interval (block size) varying from 256 B, 1 KB, 4 KB, 16 KB, to 64 KB.

As shown in Figure 13, time markers substantially improve the performance of all five workloads. Under Traffic workload, for example, enabling TSCache with a time marker interval of 4-KB improves the bandwidth by a factor of 1.7 and reduces the latency by 42.4%. Other workloads show similar results. Without time markers, we have to read the entire slab to look for the target time range, which incurs unnecessary I/Os. In the experiments, we find that TSCache without time markers can incur 16.4 times more I/Os in terms of data amount than using 4 KB-interval time markers.

The setting of time marker interval can affect performance. Using 4 KB-interval time markers, the bandwidth is improved by 16.1%–27.9% and 18.5%–26.9% and the latency is reduced by 13.9%–21.4% and 15.7%–21.6%, compared to setting the interval to 64 KB and 256 B. It indicates that either setting the interval too coarse- or too fine-grained is inefficient. The former increases the amount of unwanted data loading, while the latter increases the storage demand for storing time markers in the header part of the slab. Setting time markers at 4-KB interval is generally a proper choice.

5.4.2 Parallel Slab I/O Operations. In TSCache, each slab contains only one field of data points, which creates opportunities for us to flexibly retrieve data using multiple threads in parallel. In our prototype, we create a thread pool to efficiently allocate a thread to read or write data for each field. As Figure 14 shows, the system using parallel operations can improve the bandwidth by 5.2%–51.8% and reduce the latency by 4.8%–34.3% under the five workloads, compared to sequentially loading data in one thread.

5.5 Time-aware Caching Policy

TSCache adopts an adaptive two-level cache replacement. The cache space is divided into two partitions. The lower-level partition manages the majority data in the FIFO manner, while the upper-level partition manages "hot spot" data using LRU replacement.

In order to illustrate how the caching policy of TSCache works, we simulate a time-series data workload with "hot spots". We generate and mix two sets of queries in different time ranges of Trajectory dataset. The first set of queries accesses data points over the complete time range from Oct. 2007 to Feb. 2008. Following Latest distribution, accesses in this query set are mostly on new data points. The other query set simulates hot-spot accesses on relatively old data points, concentrated in the month of Oct. 2007. We vary the ratio of queries to new and old data points from 8:1 to 1:8 to show how TSCache responds to different workloads.

Figure 15 shows the results. We can see that TSCache outperforms both FIFO and LRU in all cases. Compared to FIFO, TSCache achieves much better performance, and this performance gap widens as the portion of accesses to old data increases. In particular, at the new/old ratio of 1:4 and 1:8, the hit ratios of TSCache are 14.2 p.p. and 17.4 p.p. higher than FIFO, respectively, which in turn results in a bandwidth increase by 37.8% and 59.1% and a latency reduction by 27.8% and 37.5%. This result is unsurprising. FIFO evicts old data first, disregarding its hotness. TSCache, in contrast, promotes old but hot data into the LRU partition, which prevents premature eviction of such hot-spot data from cache.

TSCache also performs better than LRU. LRU performs unsatisfactorily with mixed workloads, in which the new and old data compete for the limited cache space. TSCache divides the cache space into two separate partitions and turns the replacement into a scheme with two-level priorities, which effectively protects the hot data and reduces cache misses. Since each cache miss would incur a query to the backend database, which involves heavy network and storage I/O delays, removing such high-cost cache misses can bring significant performance gains. We can see that TSCache increases the bandwidth by 20.2% and decreases the latency by 17.1% compared to LRU at the new/old ratio of 1:1.

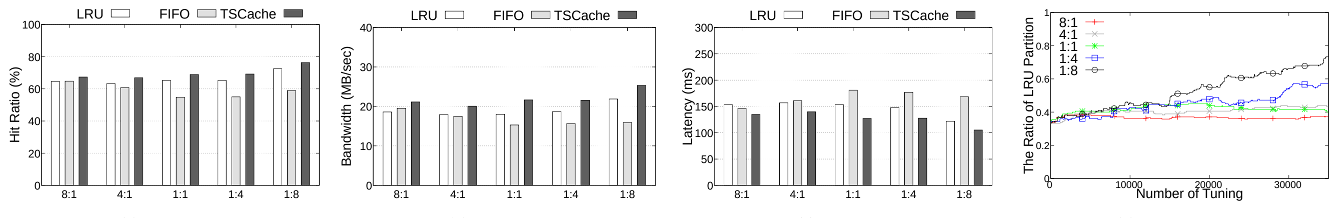
(a) Hit Ratio (b) Bandwidth (c) Latency (d) Auto-tuning

Figure 15: Performance comparison of LRU, FIFO, and TSCache replacement algorithms under Trajectory data workload.

TSCache auto-tunes its LRU and FIFO partition sizes during runtime. Figure 15(d) shows this process. Taking the workload with a ratio of 1:8 for example, the initial LRU/FIFO partition size ratio is 1:2. As cache replacement progresses, TSCache adapts to the workload and gradually increases the LRU partition size to 73%, which in effect gives more weight to the LRU-controlled partition, allowing TSCache to accommodate more old but hot data.

5.6 Compaction

In order to improve cache space utilization, a compaction process periodically runs in the background to remove duplicate data points. To evaluate its effect, we make a comparison between TSCache with disabled compaction, TSCache with normal compaction, and TSCache with optimized compaction. The normal compaction compacts slabs in the chronological order (i.e., from old to new), and the optimized compaction uses the reverse order.

Effect of compaction. Figure 16 shows the compaction performance under Trajectory workload. We can see that compaction significantly improves TSCache performance. In particular, TSCache with normal compaction increases the cache hit ratio by 10.5–36.3 p.p. across the five workloads. This is because compaction removes duplicate data points and allows us to accommodate more data in cache, which effectively improves the bandwidth by a factor of 1.2–2.6 and reduces the latency by 18.5%–59.1%.

Though simple, our optimization for compaction is effective. Due to the nature of time-series data, old data are more likely to be evicted from cache soon. Compacting slabs in the reverse chronological order (i.e., the new data first) avoids unnecessary compaction on such data. With this simple optimization, the bandwidth is further improved by 5.3%–13.2% compared to normal compaction.

Compaction triggers. TSCache initiates the compaction process when the slab node list exceeds a certain length. Although a low threshold setting minimizes duplicate data in cache, it incurs high system overhead due to the more frequently triggered compaction. We vary the threshold from 2 slabs to 64 slabs to measure the effect of different settings for compaction.

Figure 17 shows that setting this threshold either too low or too high is sub-optimal. With a low threshold setting, compaction runs too early and too frequently, which eliminates redundant data aggressively but incurs high interference to foreground queries. For example, setting the threshold to 2 slabs improves the hit ratio by only 1–3 p.p., compared to the setting of 16 slabs. However, this increase of hit ratio does not translate into performance improvement. In fact, the former's bandwidth is 0.7%–17.7% lower than the latter. In the meantime, if the threshold is set too high, the increase of data redundancy in cache also hurts performance due to the reduced hit ratio. For example, setting the threshold to 64 slabs decreases the cache hit ratio by 5.1–11.4 p.p. compared to the setting of 16 slabs, and the bandwidth decreases by 7.4%–20.7%.

Compaction overhead. We have also designed an experiment to show the overhead of compaction. To exclude the impact of caching effects, we first configure a sufficiently large cache capacity to contain the entire dataset fully in cache. Since compaction also brings benefits in keeping a short slab node list, in order to show the worst-case overhead, we arbitrarily skip the final step of updating the list structure after compaction. In other words, compaction in this experiment brings no benefit but only cost.

As shown in Figure 18, TSCache with aggressive compaction (2 slabs) decreases the bandwidth by 10.8%–12.9% and increases the latency by 12.1%–14.8%, compared to TSCache without compaction. As the threshold setting increases to 32 slabs, the less frequent compaction alleviates the bandwidth decrease and latency increase to 1.9%–7.1% and 2.1%–7.7%, respectively. We should note that this is an artificially created worst case. In experiments, we find that even simply allowing compaction to update the slab node list can offset the overhead, performing better than no compaction.

6 RELATED WORK

In recent years, time-series data has received high interest in both academia and industry [2, 26, 29, 34, 39–41, 48, 49, 53, 57, 58, 60, 67, 72]. Here we discuss the prior works most related to this paper.

Most of the prior works are on time-series databases, which are designed for different purposes and emphases [15, 18, 19, 26, 29, 34, 41, 65, 69]. For example, time-series databases for data analytics (e.g., InfluxDB) focus on optimizing data management to improve data ingestion and retrieval performance. Some time-series databases, such as Gorilla [65], are designed for data monitoring and alerting, which has a critical requirement on the response time. Some other databases, such as Graphite [18], are designed for data graphing and put more efforts on data visualization.

Another set of prior works focuses on reducing the storage overhead and increasing query performance by leveraging approximation methods, such as sampling and summarizing [33, 36, 37, 39, 49, 68]. The main purpose is to give applications quick responses without retrieving the entire dataset. SummaryStore [39], for example, is an approximate time-series data store for analytical and machine learning workloads. It aggressively compacts data streams with time-decayed summarizing based on the insight that most analysis workloads favor recent data. Agarwal et al. [36] also propose a sampling-based method to provide close-enough answers for approximate queries with estimated error bars. Cui et al. [49] propose to use time-series specific data transformations from simple sampling techniques to wavelet transformations to trade off the

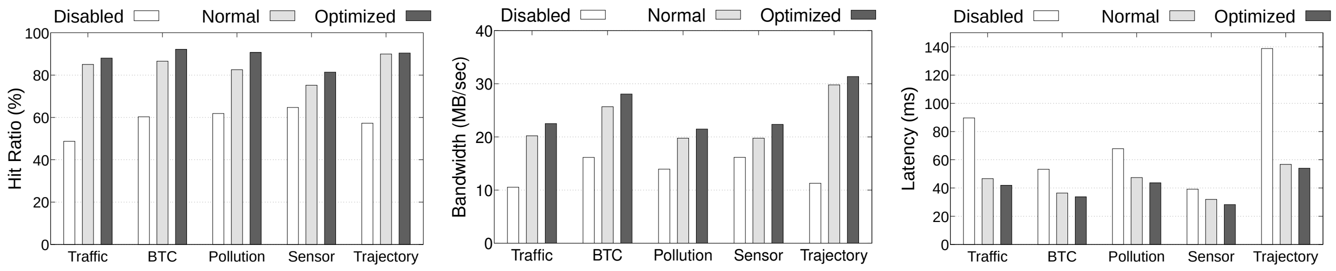

Figure 16: Hit ratio, bandwidth, and latency comparison with disabled, normal, and optimized compaction in TSCache.

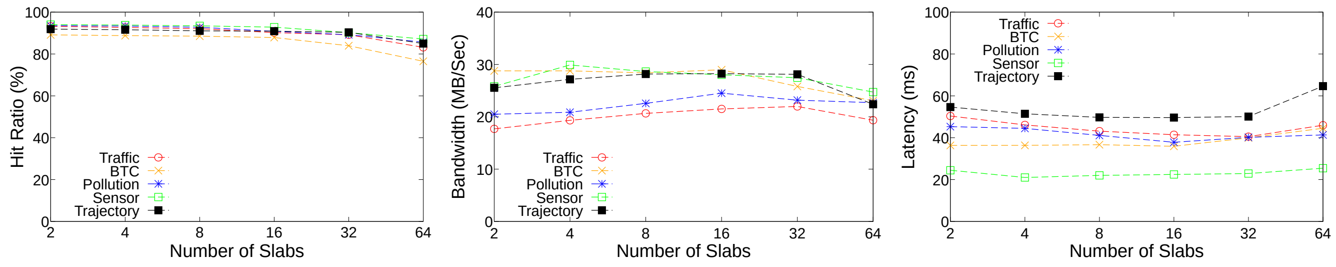

Figure 17: Hit ratio, bandwidth, and latency comparison with different compaction-triggering thresholds in TSCache.

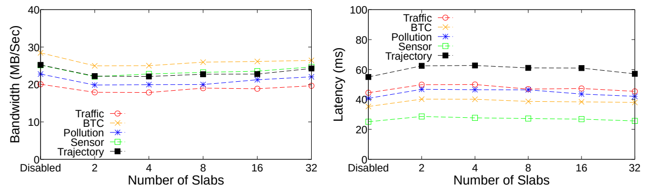

Figure 18: Compaction overhead with different thresholds.

bounded query inaccuracy for lower latency. BlinkDB [37] also allows users to run queries on data samples, which can be adaptively adjusted to make a dynamic tradeoff between accuracy and latency.

Another focus in prior works is to reduce query latency by holding data in memory [33, 35, 50, 56, 65]. To accelerate data queries for real-time analysis, Scuba [33] automatically expires data (e.g., only keep the recent data within 1~2 weeks or hours) to hold the most recent data in memory. Similarly, Gorilla [65] only stores the data during past 26 hours in memory. Shark [50] is a distributed in-memory data analysis system optimized for query speed based on temporal locality. Monarch [34] stores all data in memory to satisfy the needs for high-performance data monitoring and alerting.

Caching has been studied for improving time-series data storage system performance. Wangthammang et al. [71] use a fixed time unit to split data stream into multiple chunks, and store each chunk in Memcached for accelerating queries. Druid [73], a distributed time-series database, also splits data queries into segments to cache in local heap memory or external Memcached. Our experiments show that such an arbitrary chunking method can cause high overhead and inefficient caching. Cache replacement is a heavily studied topic. General-purpose caching algorithms, such as LIRS [55], ARC [62] and CLOCK-Pro [54], are designed for workloads in general computing environments. In contrast, our two-level caching policy is specifically tailored to time-series data. Our experimental results also show that it is important to exploit the unique properties of time-series data workloads in the cache system design.

Data compression has also been studied as an effective method to improve query performance on time-series data. Yu et al. [74] propose to use reinforcement learning to handle dynamic data patterns thereby adjusting the compression scheme at a fine granularity, thus increasing compression ratio and decreasing overhead. Gorilla [65] adopts delta-of-delta compression on timestamps and XOR compression on data value for memory efficiency. Sprintz [42] focuses on optimizing compression on integers for time-series data points. Succinct [35] proposes an efficient Compressed Suffix Arrays technique to accommodate more compressed data in memory, and it can execute queries directly on compressed data representation, thereby avoiding the overhead of decompressing data.

Compared to these prior works, TSCache focuses on providing an effective and efficient flash-based caching solution to enhance user experience with time-series databases. Our experiments and studies show that with careful design, this solution can provide large-capacity, high-speed caching services at low cost.

7 CONCLUSION

In this paper, we present a highly efficient flash-based caching solution, called *TSCache*, to accelerate database queries for time-series data workloads. We have designed a set of optimization schemes by exploiting the unique properties of time-series data. We have developed a prototype based on Twitter's Fatcache. Our experimental results demonstrate the effectiveness and efficiency of the proposed cache system design.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and insightful comments. This work was supported in part by the U.S. National Science Foundation under Grant CCF-1910958.

REFERENCES

[1] 1995. SHA-1. https://en.wikipedia.org/wiki/SHA-1.
[2] 2000. Fintime. https://cs.nyu.edu/cs/faculty/shasha/fintime.html.
[3] 2000. Time Series in Finance: the array database approach. https://cs.nyu.edu/shasha/papers/jagtalk.html.
[4] 2004. Intel lab data. http://db.csail.mit.edu/labdata/labdata.html.
[5] 2009. cJSON. https://github.com/DaveGamble/cJSON.
[6] 2010. Facebook Flashcache. https://github.com/facebookarchive/flashcache.
[7] 2010. YCSB-TS. http://tsdbbench.github.io/YCSB-TS/.
[8] 2011. libMemcached. https://libmemcached.org/libMemcached.html.
[9] 2012. Air pollution index. https://en.wikipedia.org/wiki/Air_Pollution_Index.
[10] 2013. Fatcache. https://github.com/twitter/fatcache.
[11] 2013. How Twitter monitors millions of time series. https://www.oreilly.com/content/how-twitter-monitors-millions-of-time-series/.
[12] 2013. Introducing Kale. https://codeascraft.com/2013/06/11/introducing-kale/.
[13] 2014. Pollution data. http://iot.ee.surrey.ac.uk:8080/datasets.html#pollution.
[14] 2014. Road traffic data. http://iot.ee.surrey.ac.uk:8080/datasets.html#traffic.
[15] 2019. Introducing AresDB: Uber's GPU-powered open source, real-time analytics engine. https://eng.uber.com/aresdb/.
[16] 2020. Bitstamp exchange data. https://www.cryptodatadownload.com/data/bitstamp/.
[17] 2021. DB-Engines ranking of time series DBMS. https://db-engines.com/en/ranking/time+series+dbms.
[18] 2021. Graphite. https://graphiteapp.org/.
[19] 2021. InfluxDB. https://www.influxdata.com/.
[20] 2021. InfluxDB design insights and tradeoffs. https://docs.influxdata.com/influxdb/v1.7/concepts/insights_tradeoffs/.
[21] 2021. InfluxDB key concepts. https://docs.influxdata.com/influxdb/v1.7/concepts/key_concepts/#measurement.
[22] 2021. influxdb1-clientv2. https://github.com/influxdata/influxdb1-client.
[23] 2021. JSON. https://www.json.org.
[24] 2021. LevelDB. https://github.com/google/leveldb.
[25] 2021. Memcached. https://github.com/memcached/memcached.
[26] 2021. OpenTSDB. http://opentsdb.net/.
[27] 2021. Redis. https://github.com/redis/redis.
[28] 2021. Time series database (TSDB) explained. https://www.influxdata.com/time-series-database/.
[29] 2021. TimescaleDB. https://www.timescale.com/.
[30] 2021. Trend of the last 24 months. https://db-engines.com/en/ranking_categories#chart-container-line-24.
[31] 2021. What Is a time series? https://www.investopedia.com/terms/t/timeseries.asp.
[32] 2021. What is time series data? https://www.influxdata.com/what-is-time-series-data/.
[33] Lior Abraham, John Allen, and et al. 2013. Scuba: Diving into data at Facebook. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1057–1067.
[34] Colin Adams, Luis Alonso, and et al. 2020. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3181–3194.
[35] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. Oakland, CA, USA, 337–350.
[36] Sameer Agarwal, Henry Milner, and et al. 2014. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD'14)*. Snowbird, UT, USA, 481–492.
[37] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. Prague, Czech Republic, 29–42.
[38] Nitin Agrawal, Vijayan Prabhakaran, and et al. 2008. Design tradeoffs for SSD performance. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'08)*. Boston, Massachusetts, USA, 57–70.
[39] 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)*. Shanghai, China, 647–664.
[40] Shadi Aljawarneh, Vangipuram Radhakrishna, Puligadda Veereswara Kumar, and Vinjamuri Janaki. 2016. A similarity measure for temporal pattern discovery in time series data generated by IoT. In *In 2016 International conference on engineering MIS (ICEMIS'16)*. Agadir, Morocco, 1–4.
[41] Michael P Andersen and David E. Culler. 2016. BTrDB: Optimizing storage system design for timeseries processing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. Santa Clara, CA, USA, 39–52.
[42] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the Internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.
[43] Simona Boboila and Peter Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. San Jose, CA, USA, 115–128.
[44] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory based solid state drives. *ACM Transactions on Storage* 12, 3 (June 2016), 13:1–13:39.
[45] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of ACM SIGMETRICS (SIGMETRICS'09)*. Seattle, WA, USA, 181–192.
[46] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. Tucson, Arizona, USA, 22–32.
[47] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-Speed data processing. In *the 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. San Antonio, TX, USA, 266–277.
[48] Tak chung Fu. 2011. A review on time series data mining. *Engineering Applications of Artificial Intelligence* 24, 1 (2011), 164–181.
[49] Henggang Cui, Kimberly Keeton, and et al. 2015. Using data transformations for low-latency time series analysis. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SOCC'15)*. Hawaii, USA, 395–407.
[50] Cliff Engle, Antonio Lupher, and et al. 2012. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. Scottsdale, Arizona, USA, 689–692.
[51] Philippe Esling and Carlos Agon. 2012. Time-series data mining. *Comput. Surveys* 45, 1 (2012), 1–34.
[52] Laura M. Grupp, Adrian M. Caulfield, and et al. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. New York, NY, USA, 24–33.
[53] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.
[54] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX ATC'05)*. Anaheim, CA, USA, 323–336.
[55] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceeding of ACM SIGMETRICS (SIGMETRICS'02)*. Marina Del Rey, CA, USA, 31–42.
[56] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTPOLAP main memory database system based on virtual memory snapshots. In *In IEEE 27th International Conference on Data Engineering (ICDE'11)*. Hannover, Germany, 195–206.
[57] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. Boston, MA, USA, 421–436.
[58] Sanjeev Kulkarni, Nikunj Bhagat, and et al. 2015. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Melbourne, Victoria, Australia, 239–250.
[59] Nitin Kumar, Venkata Nishanth Lolla, and et al. 2005. Time-series bitmaps: A practical visualization tool for working with large time series databases. In *Proceedings of the 2005 SIAM international conference on data mining (ICDM'05)*. Houston, TX, USA, 531–535.
[60] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web (WWW'10)*. Raleigh, NC, USA, 661–670.
[61] Jian Liu, Kefei Wang, and Feng Chen. 2019. Reo: Enhancing reliability and efficiency of object-based flash caching. In *the 39th International Conference on Distributed Computing Systems (ICDCS'19)*. Dallas, TX, USA, 578–588.
[62] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage (FAST'03)*. San Francisco, CA, 115–130.
[63] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and InfluxDB. In *Studienarbeit, Université Libre de Bruxelles*.
[64] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.
[65] Tuomas Pelkonen, Scott Franklin, and et al. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.
[66] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[67] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. 2014. Aggregation and degradation in jetstream: Streaming analytics in the wide

area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA, USA, 275–288.

[68] Galen Reeves, Jie Liu, Suman Nath, and Feng Zhao. 2009. Managing massive time series streams with multi-scale compressed trickles. *Proceedings of the VLDB Endowment* 2, 1 (2009), 97–108.

[69] Alexander Visheratin, Alexey Struckov, and et al. 2020. Peregreen–modular database for efficient storage of historical time series in cloud environments. In *Proceedings of 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 589–601.

[70] Kefei Wang and Feng Chen. 2018. Cascade mapping: Optimizing memory efficiency for flash-based key-value caching. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'18)*. Carlsbad, CA, USA, 464–476.

[71] Thada Wangthammang and Pichaya Tandayya. 2018. A software cache mechanism for reducing the OpenTSDB query time. In *the 18th International Symposium on Communications and Information Technologies (ISCIT'18)*. Bangkok, Thailand, 60–65.

[72] Xiaomin Xu, Sheng Huang, and et al. 2014. TSAaaS: Time series analytics as a service on IoT. In *2014 IEEE International Conference on Web Services (ICWS'14)*. Anchorage, AK, USA, 249–256.

[73] Fangjin Yang, Eric Tschetter, and et al. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD'14)*. Snowbird, UT, USA.

[74] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-Level data compression using machine learning in time series database. In *the 36th International Conference on Data Engineering (ICDE'20)*. Dallas, TX, USA, 1333–1344.

[75] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge Discovery and Data mining (KDD'11)*. San Diego, CA, USA, 316–324.

[76] Jing Yuan, Yu Zheng, Chengyang Zhang, and et al. 2010. T-drive: Driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'10)*. San Jose, CA, USA, 99–108.