IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 30, 2019, accepted September 21, 2019, date of publication September 26, 2019, date of current version October 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2943876

EdgeDB: An Efficient Time-Series Database for Edge Computing

YANG YANG[1], QIANG CAO[1], (Senior Member, IEEE), AND HONG JIANG[2], (Fellow, IEEE)

[1]Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Ministry of Education, Huazhong University of Science and Technology, Wuhan 430074, China

[2]Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA

Corresponding author: Qiang Cao (caoqiang@hust.edu.cn)

This work was supported in part by the Creative Research Group Project of NSFC under Grant 61821003, in part by the NSFC under Grant 61872156, in part by the National Key Research and Development Program of China under Grant2018YFA0701804, in part by the Fundamental Research Funds for the Central Universities under Grant 2018KFYXKJC037, in part by the U.S. NSF under Grant CCF-1704504 and Grant CCF-1629625, and in part by the Alibaba Group through Alibaba Innovative Research (AIR) Program.

ABSTRACT Massive time-series data streams from high-sampling-frequency sensors in Internet of Things (IoT) can overwhelm the networks connecting the sensors to centralized clouds. Thus, edge computing servers have to be introduced to locally store and analyze growing time-series data. Unfortunately, conventional time-series databases exhibit low efficiency on edge nodes with limited resources for both computation and storage. In this paper, we propose a highly efficient time-series database, called EdgeDB, to fully utilize the capacity of the edge nodes. EdgeDB effectively improves the performances of both inserting and retrieving data from ingest streams by efficiently merging multiple streams and optimizing the storage data structure concurrently. EdgeDB first compactly organizes multiple online streams into a tablet within a time window and embeds predefined aggregate query results together. EdgeDB adopts Time Partitioned Elastic Index (TPEI) to build indexing on all tablets, enhancing the time-range query performance while reducing the memory usage by optimizing the indexing storage. EdgeDB further develops Time Merged Tree (TMTree) to combine a set of tablets into a large one, significantly boosting the write throughput and potentially strengthening the performance of inter-tablet query. Extensive experiments based on real-world datasets show that, compared with the state-of-the-art time-series database BTrDB, EdgeDB achieves performance improvements of up to 2.2× in insert throughput, 3.6× in write throughput, and 67% in query latency with lower memory consumption.

INDEX TERMS Time-series database, edge computing, time partitioned elastic index, I/O optimization.

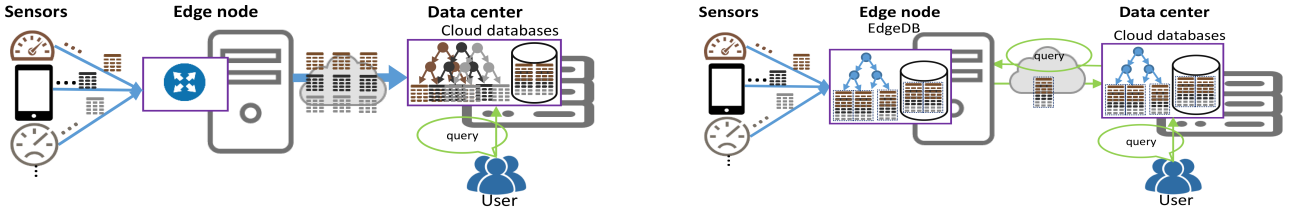
I. INTRODUCTION

Internet of Things (IoT) applications, commonly deploying a myriad of sensors to collect a large amount of time-series data from physical environments around human beings, are designed to help us monitor [22], [31], analyze [10], [34], forecast our concerned events [14], [28], and then make timely and correct responses.

With the explosive growth of time-series data generated by high-sampling-frequency sensors, the long-distance networks between sensors and data centers have become a performance bottleneck for traditional data management systems running in the datacenter-based cloud environment, as illustrated in Figure 1a, due to high data transfer overhead [3]. This necessitates IoT infrastructures to embrace edge computing that locally processes the data on edge nodes efficiently, in order to improve the responsiveness and prevent sensor-generated data from flooding the centralized cloud.

Although existing databases have the ability to process large-scale time-series data streams on cloud-based infrastructure, they are not well designed to run on the edge nodes with limited hardware resources, power budget, and scalability. Existing time-series databases [2], [12], [22] separately ingest, organize, store, and query data of each stream in a rational table. And they provide standard inter-table operations to enable inter-stream data retrieval and processing, but at very high resource cost. This limitation can be overcome by scaling resources up and/or out to ensure high overall performance, a solution that is clearly not suitable nor applicable for emerging edge nodes with limited hardware resources.

The associate editor coordinating the review of this manuscript and approving it for publication was Shaojun Wang.



(a) A cloud-centralized framework of data management systems for IoT.

(b) An edge-database based framework of data management systems for IoT.

FIGURE 1. A comparison of cloud-centralized framework vs edge-database based framework for IoT. Cloud-centralized framework needs to transfer all time-series data to remote cloud for data management with significant network overhead. Edge-database based framework can utilize the edge node to locally process the collected data, and transfers important data and aggregated values to cloud, then collaborates with the centralized databases to perform detailed queries.

Nevertheless, it is desirable for the edge nodes to accommodate not only high-throughput data ingestion but also real-time data retrieval from both fresh and historical data in local storage to satisfy the requirements of IoT applications.

In this paper, we present EdgeDB, an efficient time-series database with edge servers (shown in Figure 1b) designed to manage thousands of high-sampling-frequency sensors while providing higher insert performance, query performance, and write speed, with lower resource requirement than existing databases. The key idea behind EdgeDB is to design an optimal organization and process flow by efficiently combining multiple online streams in both query-friendly and store-friendly ways, enabled by the three key mechanisms of multi-stream merging, indexing, and storing. The multi-stream merging mechanism is designed to compactly re-organizes multiple correlated data streams within a time window into a tablet. Then, the multi-stream indexing mechanism, called Time Partitioned Elastic Index (TPEI), is proposed to index these tablets efficiently, enabling real-time queries. Finally, a write-optimized strategy is developed to combine multiple tablets into a group to allow fast inter-stream accesses.

In summary, the contributions of this paper are as follows:

1) We propose a multi-stream merging mechanism to compactly organize multiple correlated streams together at runtime, supporting highly efficient insertion and join query operations; and introduce Time Partitioned Elastic Index to accelerate time-range queries with small memory overheads.
2) We present Time Merged Tree to merge multiple tablets into a large group to flush to the storage devices with a single write operation, to improve the write throughput and to speed up inter-tablet join query operations.
3) We implement and evaluate the EdgeDB prototype. The experimental results driven by real-world datasets demonstrate that EdgeDB outperforms a state-of-the-art time-series database BTrDB by up to 3.6× in write throughput, 2.2× in insert throughput, and up to 67% in query latency, with lower memory consumption.

The remainder of this paper is organized as follows: Section II presents background and motivation. The design details of EdgeDB are described in Section III. In Section IV, we evaluate EdgeDB with experimental results. Section V concludes this paper.

II. BACKGROUND AND MOTIVATION

A. EDGE COMPUTING OF IoT

Internet of Things (IoT) is poised to fundamentally change how we interact with our surrounding environments [31], [35]. By deploying a variety of increasing numbers of sensors with high-sampling-frequency, we are able to perceive the surrounding environment quickly and clearly. More importantly, we can make informed and timely decisions in emergency situations using the sensor-generated data. However, the considerable volumes of data make the long-distance networks a severe performance bottleneck, as illustrated in Figure 1a, because of a long time required for data transfer to remote servers. For example, a small electricity grid with 1,000 smart meters will produce 22.4 MB data per second, and the data must be transferred to datacenters over intermittent LTE with high transmission delays [2]. Even though these servers have powerful hardware, the high transfer latency makes it difficult, if not impossible, for the cloud to make real-time decisions based on these data. To overcome this problem, edge computing is introduced to provide near-data processing [15], [33].

With the help of edge computing, we have the potential to store and process all collected data in a timely manner on the edge nodes, instead of remote servers. For example, an autonomous vehicle generates gigabytes of data every second [6] that require real-time processing to make correct decisions under various circumstances. The vehicle-mounted computer is a typical edge node, which can manage and analyze these data locally, and provide quick responses or advance warnings for the driver.

Therefore, edge nodes should be able to support extremely high insertion throughput, as well as real-time responses to various types of queries. For instance, users not only need to query the raw data tuples of any time-series stream to analyze detailed phenomena, but also need to get the time-range based aggregated values to see the holistic views, or perform join query to get the correlated data tuples from a number of streams within a period of time to further comprehensively analyze these streams. As a concrete example, in order to calculate the Pearson Correlation Coefficients among all streams within a (T1, T2) time period, we need to first perform "*Select * from all streams where Time>T1 & Time<T2*" to get the data of these streams for further processing.

However, the edge nodes generally have limited compute and storage resources partly due to power and size constraint. Therefore, edge nodes with common desktop or server configurations have to sufficiently exploit their potentials to process and manage tremendous volumes of time-series data from sensors with a highly efficient time-series database.

B. EXISTING DATABASES

Traditionally, mainstream databases have been employed to manage time-series data, such as Redis [25], Cassandra [4], MySQL [21], and VoltDB [27]. These databases are generic and full-feature data management systems, and use a single big rational table to organize all time-series streams. Each row in the table represents a sampling data record from a stream at a given timestamp. They can support a variety of simple and complex queries. However, these generic databases exhibit relatively low insertion performance for highly intensive time-series streams due to various sophisticated mechanisms to support ACID transactions, which might not be necessary for time-series data. Previous study [24] reveals that Cassandra provides the highest request-processing throughput among these databases, but can only insert 19k records per second at a single processing node. Therefore, they can only satisfy IoT applications with small-scale low-sampling-frequency sensors for simple data analytics.

With the development in IoT technologies and big data applications, the need to understand some transient and instantaneous phenomena has resulted in high-sampling-frequency sensors being increasingly deployed to collect massive time-series data in many key application scenarios including smart power grids [5], building systems [11], industrial processes [9], etc. Therefore, some dedicated time-series databases have been developed to provide adequate performance for these applications in recent years, such as Gorilla [22], BTrDB [2] and InfluxDB [12]. Considering that different streams are relatively independent, these databases organize each data stream separately, and insert every stream into an individual schema, where one column only contains the defined field of the corresponding stream. These time-series databases share a common workflow in which data are *separately imported, organized, stored, and queried, in an independent table for each time-series data stream*, referred to as the *single-stream* data organization. They can perform standard inter-table operations to process inter-stream data, but at a very high resource cost. Therefore, they are usually designed to run on high-performance servers or scalable clusters, and are not suitable for resource-constrained edge nodes with low/mid configurations such as desktops or low-end servers. In summary, the design principles of these databases limit their applicability to the edge nodes due to the inappropriate data organization, inefficient index structures, and lack of write optimizations. Next, we will analyze these three limitations in existing dedicated time-series databases.

1) DATA ORGANIZATION

The *single-stream* organization exhibits low efficiency due to its independent management of each stream. With such organization, the system inserts, indexes, and writes each data stream separately, resulting in high resource consumption and I/O overhead when processing large-scale time-series data streams. Furthermore, any query involving multiple streams, needs one or more expensive join operations that entail reading from disjoint tables associated with these multiple streams in a random-access pattern, leading to exceedingly long response time. As a result, the *single-stream* organization is not an optimal choice for edge nodes.

2) WRITING MECHANISM

The incoming time-series data need to be persisted onto storage devices as soon as possible, because edge nodes generally have limited memory and power, in addition to being vulnerable to accidental crashes. However, such persistency requirement results in a large number of small writes for the collected data, which can become a performance bottleneck for the edge nodes owing to limited storage resources. Existing works [22], [26], implicitly or explicitly assuming reliance on high-performance distributed storage systems, pay little attention to this potentially serious write problem.

BTrDB [2] randomly merges data to be written into a batch in order to flush them with a single write operation. This merge strategy, however, ignores the correlation among data sources and the time ranges of merged data, resulting in disordered data layout on disks and thus significantly degrading future query performance.

LSM-tree is a write-optimized structure for key/value stores, and consists of a number of indexed components with an exponentially increasing size with the levels. LSM-tree-based databases [4], [12] periodically write the in-memory components of the time-series data to storage devices in order to compact them with their on-disk counterparts. Consequently, these databases achieve high write throughput, but at the expense of significant read and write amplifications [18], [19].

3) INDEXING MECHANISM

The databases heavily depend on an indexing mechanism to quickly serve query and analysis on massive time-series data.

B+ tree index is one of the most widely used indexing structure in existing databases, such as MySQL [21] and Waterwheel [32]. For the continuously incoming time-series data streams, B+ tree needs to frequently split index nodes to create substantial branches to index them, resulting in excessive overhead of node splits, and accumulates into a large index tree structure gradually, which incurs significant performance overhead when querying.

Emerging time-series databases primarily focus on the time-range query, instead of value-range query, for time-series data. Kairosdb, OpenTSDB, and InfluxDB adopt LSM-tree [20] or its variants to index time-series data in

the order of timestamps. However, the unavoidable data compaction limits their performance under continuous data ingestion.

BTrDB uses TPTree [2] to index the collected time-series data based on timestamps, and to record the aggregated values of the data in internal index nodes to speed up the aggregation query. These index data take up large memory space. Although the indexes can be stored on disks, their query performance is significantly degraded, because the traditional query algorithm requires traversing the TPTree from the root to leaf nodes, and fetching multiple unnecessary internal nodes in a random access pattern.

LittleTable [26] and Gorilla adopt a mapping table to record start-times and addresses of all tablet blocks. This index structure works well for range query, but is inefficient for aggregation query and join query.

C. MOTIVATION

Existing time-series databases cannot adequately exploit the potentials offered by the underlying hardware to maximize the performances of both data organization and analysis in real-time as expected by IoT applications. This inefficiency can be mitigated by scaling the underlying hardware in data-centers with thousands of powerful servers and large memory. However, this is hardly possible in a typical IoT environment where time-series data streams are to be stored and processed locally by single edge node with limited resources, scale and power.

Thus, we are motivated to design a novel dedicated time-series database, EdgeDB, to ingest, store and query massive time-series data streams efficiently on edge nodes. The database is desired to have several necessary capabilities, including high insertion throughput, high write performance, low query response time, and low resource requirement. In addition, the database should be able to effectively cooperate with the large-scale databases in the cloud to process these data.

EdgeDB is designed to effectively merge multiple time-series streams on front-end edge nodes by efficiently organizing and storing correlated data streams. EdgeDB further optimizes the indexing and writing processes for multiple streams by designing I/O friendly data structures. Data and indexes from the optimized data and index organizations can be delivered to the centralized databases in the cloud efficiently for further analysis. Additionally, EdgeDB can also perform queries on local data, effectively collaborating with the centralized databases to perform global queries.

III. EDGEDB DESIGN

Figure 2 shows the architecture of EdgeDB, which consists of a data merging module, an indexing module, and a data storing module. When receiving a set of sampling data from sensors in a given time window, the data merging module re-organizes the dataset into a tablet for each stream group. Then, the indexing module builds Time Partitioned Elastic Indexes (TPEIs) on these tablets of each stream group,

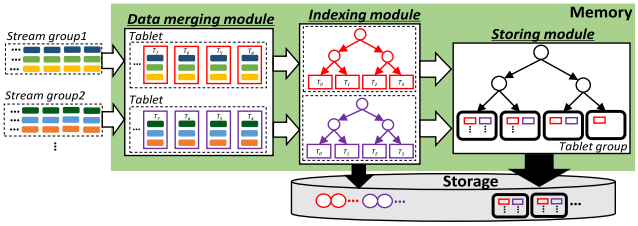


FIGURE 2. The architecture of EdgeDB.

and periodically writes the index data to disks for persistency. The data storing module constructs Time Merged Tree (TMTree) to combine tablets of different stream groups into larger tablet groups, which can be flushed onto disks with large writes, thus boosting the write throughput.

A. MERGING: MULTI-STREAM DATA ORGANIZATION

In order to overcome the drawbacks of the *single-stream* data organization, EdgeDB merges multiple correlated streams from their corresponding sensors and pre-processes them into a stream group. First, EdgeDB splits a time-series data stream into a sequence of shards by timestamp within a time window, such as 2^t-ns window where t is pre-defined. Second, the shards of different streams belonged to a stream group but within the same time range are merged into a tablet. The main purpose of the merge strategy is to enable the correlated streams to be kept together to process in batches. The correlation can be defined based on properties with regard to, e.g., sensor location, type, owners, etc. We expect the frequently-queried streams to be merged in a stream group. Likewise, users can define a specific-correlation among streams. The number of the merged streams can also be dynamically adjusted during runtime according to the change of properties.

Once the correlated streams and the time windows are determined, EdgeDB can construct these tablets with received data, each of which contains the shards of these streams, and every shard has a set of rows. Each stream has its own table schema which defines a set of columns (i.e., fields), each of which has a name, type, and default value. And each row is a sampling record, and keeps a data tuple with any number of key fields and data fields collected by a sensor at a certain point in time, and the timestamp is defined as the primary key. The rows in a shard are stored in chronological order.

All shards from different streams in a stream group are successively placed in the tablet space. The tablet records the start addresses and offsets of its constituent shards in a tablet descriptor, which is located in the head of the tablet. The descriptor also records a set of pre-defined aggregated values from these correlated shards, such as the maximal values among the set of streams in the time window. These aggregated values can be easily computed once all shards are gotten available. Users can also define their concerned aggregated values. More detailed aggregated values also can be recorded in the index, as described below in Section III-B.

Additionally, the time-series data reduction techniques, such as compression [13], [22], approximation [16], [23], can be further employed to reduce the size of shards. And the tablet descriptor can record the corresponding metadata of these approaches.

Merging multiple small shards into a large tablet can greatly improve insert performance owing to fewer processes and I/O operations. Meanwhile, we record the raw data tuples and the aggregated values of these streams in the same tablets, potentially boosting the join query performance for the correlated streams. When there is an intra-tablet join query for several correlated streams in a tablet, EdgeDB only needs to search on an index to locate the addresses of queried data, and obtains the requested data by performing a single read operation. Even if a query is not hit on those pre-prepared aggregated values, and needs to retrieve from raw data, it still benefits from the data layout of merged correlated streams.

B. INDEXING: TIME-PARTITIONED ELASTIC INDEX

In order to speed up the time-range query and reduce the memory usage of index data, we propose a Time-Partitioned Elastic Index (TPEI) based on Time Partitioned K-ary Tree (TPTree).

Time Partitioned K-ary Tree (TPTree) is first introduced by BTrDB [2]. Its every leaf node contains a set of the original data tuples (i.e., a tablet) collected within a 2^t-ns time window; every internal index node records the addresses and aggregated values (e.g., min, mean, max, count, and standard deviation, etc.) of its child nodes, and its time window (power-of-two ns) is equal to the sum of time windows of the child nodes. The top part of Figure 3 illustrates a Time Partitioned 2-ary Tree with 6 leaf nodes (tablets) and 6 internal nodes. Each internal node indexes two child nodes and records aggregated values about its child nodes to support efficient aggregation queries.

Upon receiving the tablets from the data merging module, we compute the aggregated values within a stream, or among multiple streams, then insert index these tablets into TPTree as leaf nodes, according to their time ranges. Finally, we repeatedly update the aggregated values of the corresponding upper internal nodes until the root of the TPTree is reached.

However, TPTree can consume a significant amount of memory space with the continuous influx of new data tuples. For example, the size of index data for a single stream collected from a smart grid meter in a year is up to 20GB, making it almost impossible to keep all index data in memory, especially for the resource-constrained edge nodes. As a result, long-running databases have to flush the index data to the storage devices to make room for incoming data. This means that a query needing to retrieve the historical time-series data that are not likely to be hit in the in-memory index, must traverse on-disk TPTrees from the root to leaf nodes to read intermediate index nodes from disks, generating multiple random read I/Os. Hence, this query procedure is very inefficient for the historical data. To save memory space while enabling real-time queries for both fresh and historical data, we propose a new TPEI with a novel query procedure.

First, to avoid unnecessary memory usage in edge nodes, we design a TPEI containing multiple TPTrees with different time granularity, including at most 7 daily TPTrees ($TPTree_d$), at most 5 weekly TPTrees ($TPTree_w$), at most 12 monthly TPTrees ($TPTree_m$), and a global TPtree ($TPTree_g$). $TPTree_d$ indexes the tablets collected every day within the latest week, $TPTree_w$ indexes the tablets collected every week within the latest month except the tablets in $TPTree_d$s; $TPTree_m$ indexes the tablets collected every month of the latest year, but does not index the tablets collected within the latest month; $TPTree_g$ indexes all tablets collected before the latest year. Because time-series data tuples come into the system in chronological order, EdgeDB only needs to hold the latest TPTree_d in memory to build index for new incoming data tuples. All previous TPTrees will never be updated again, and can be persisted on disks to reduce the memory usage of the index.

Furthermore, to support timely query for the historical data, we design a novel query procedure for on-disk TPTree. We observe that, according to the queried time range, we can directly locate the target index nodes that contain the queried aggregated results or the addresses of the queried data tuples, without traversing the intermediate index nodes between the root and the target index nodes, reducing unnecessary read I/Os. For each on-disk TPTree, it has the determined structure information, which includes the node layout, the number of nodes in each layer, its number of branches (which is K in K-ary), its height, the window length of its leaf node (2^t-ns), the time range covered by the TPTree (i.e., the time window of the root node). Moreover, according to the time window of any parent node, we can calculate the time window of each of its child node, and its branch ID (branch_{ch_pa}) in the parent node. Further, given the branch ID of the parent node (branch_{pa_TP}) in the TPTree, that of its child node (branch_{ch_TP}) in the TPTree can be calculated using the following formula:

$$branch_{ch\_TP} = branch_{pa\_TP} * K + branch_{ch\_pa}.$$

The layer ID (layer_{ch_TP}) of the child node can also be obtained based on that of its parent node (layer_{pa_TP}) as follows:

$$layer_{ch\_TP} = layer_{pa\_TP} + 1.$$

The branch ID and layer ID of each node are referred to as its position information in the corresponding TPTree, and thus can be calculated from the position of its parent.

Therefore, based on the above features of TPTree, for a time-range query, we can calculate the position information of the target index nodes as follows. First, we use the start time and time window of the TPTree's root node to calculate the start times and time windows of its child nodes. We compare the queried time range with the time windows of these child nodes, then determine which subtrees to continue retrieving, and calculate the position information of the

corresponding child nodes in the subtrees. This process is repeated for the subtrees until the queried target index nodes that contain the requested aggregated values are found, or the lowest-level internal nodes that directly point to the requested data are reached, and finally yields the position information of the target index nodes in the TPTree. EdgeDB can further calculate the offsets (called *nodeoffs*) of the target index nodes within all internal nodes of the TPTree using the position information as follows:

$$nodeoff = \sum_{n=1}^{layer-1} count_n + branch_{ch\_TP},$$

where *layer* represents the $layer_{ch\_TP}$ of the node, and $count_n$ represents the number of nodes in the nth layer of the TPTree. Next, we need to determine the storage addresses of these target index nodes using the *nodeoffs*.

For the TPTree, each internal node contains the same data structure with the same data size. We can store all index nodes of each TPTree into an identical file. Each node is stored in order of its layer into the corresponding file, and all index nodes in a layer are contiguously stored (see the bottom of Figure 3). According to such a storage layout, EdgeDB can calculate the file offsets by multiplying their *nodeoffs* with the size of the internal node, then fetch these target index nodes from disks, avoiding reading unnecessary intermediate nodes. The pseudo code for this query procedure is shown in Algorithm 1. The algorithm starts by getting the in-memory structure information of the queried TPTree (line 2), then calculates the position information of the target index nodes (line 4-18). Finally, the algorithm obtains their offsets within the corresponding files according to the position informantion, and reads the nodes from disks (line19-25). In summary, for any on-disk TPTree, EdgeDB only maintains the TPTree's structure information and corresponding file path (collectively referred to as **TPTree_attr** henceforth) in memory, and can directly locate the storage address of any index node in the TPTree.

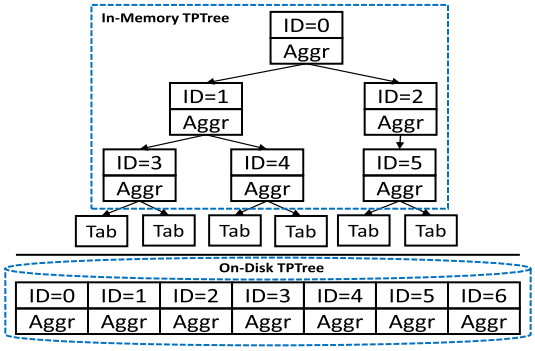


FIGURE 3. An example of a TPEI consisting of an in-memory TPTree and an on-disk TPTree. These two 3-layer TPTrees each contain 6 internal nodes and 7 internal nodes, respectively.

At the beginning of a day, EdgeDB creates a new $TPTree_d$ to index new incoming data tuples. Meanwhile, the system flushes the previous $TPTree_d$ into a new file, and maintains its **TPTree_attr** in memory. Furthermore, in order to avoid excessive TPTrees in this index structure, EdgeDB combines on-disk TPTrees periodically (i.e., every week, every month, and every year). For example, when a new week comes, EdgeDB combines all last seven $TPTree_d$s into a new $TPTree_w$, and stores the $TPTree_w$ to disks, then reclaims these old $TPTree_d$s. The combining operation is simple and lightweight. For two TPTrees with adjacent time ranges, we only need to combine the rightmost node in the previous TPree with the leftmost node in the following TPTree at each layer into a new node, from bottom to up.

In summary, an in-memory $TPTree_d$ only takes at most 53MB space. We can further reduce the memory usage of the index by adding finer-grained TPTrees (e.g., hourly TPTrees) into TPEI. In addition, with the in-memory $TPTree_d$, we can efficiently index incoming time-series data, and update the aggregated values of the upper internal nodes from bottom to top, and finally respond to user requests with these

Algorithm 1 Query Algorithm With On-Disk TPTree

Input: *Trange*: queried time range; *TPTree_attr*; *Nodesize*: the size of node;
Output: LeafNodes[]: leaf nodes within Trange;

1: Initialize Indexnodes[][], height ← 0, pnode, cnode;
2: cnode.timewindow ← TPTree_attr.timerange, cnode.$branch_{ch\_TP}$ ← 0, cnode.$layer_{ch\_TP}$ ← 0;
3: Indexnodes[0][] ← cnode;
4: **for** pnode ← each node in Indexnodes[height] **do**
5: **for** cnode ← each child node of pnode **do**
6: cnode.timewindow ← ComputeWindow(pnode.timewindow);
7: **if** *Trange* overlaps with *cnode.timewindow* **then**
8: /∗ calculate on this cnode (subtree)∗/
9: cnode.$branch_{ch\_TP}$ ← pnode.$branch_{ch\_TP}$*K + $branch_{ch\_pa}$;
10: cnode.$layer_{ch\_TP}$ ← pnode.$layer_{ch\_TP}$ + 1;
11: Indexnodes[height+1][] ← cnode;
12: **end if**
13: **end for**
14: height ← height+1;
15: **if** height ≥ TPTree_attr.height **then**
16: break; /∗ search until the lowest index nodes∗/
17: **end if**
18: **end for**
19: **for** pnode ← each node in Indexnodes[height] **do**
20: nodeoff ← GetOffset(pnode.$branch_{ch\_TP}$, pnode.$layer_{ch\_TP}$);
21: offset ← nodeoff * Nodesize;
22: indexnode ← ReadIndexNode(TPTree_attr. filename, offset);
23: address ← GetAddress(indexnode);
24: LeafNodes[] ← ReadLeafNode(address);
25: **end for**
26: return LeafNodes;

results in a timely manner. More importantly, for the on-disk TPTrees, EdgeDB only needs a single read operation to fetch a target index node, resulting in better query performance.

C. STORING: WRITE OPTIMIZATION

In practice, an IoT system with thousands of high-sampling-frequency sensors can generate hundreds of, even thousands of tablets into the memory of an edge node every second [2]. Edge nodes generally are required to fast write these tablets into disks, avoiding memory out-of-space. Typically, the size of a tablet is limited to a few hundreds of KBs. In this scenario, the time-series databases running on edge nodes generate a large number of small writes every second. By testing this write process in EdgeDB with a single disk, we find that the disk utilization (the fraction of time with busy periods) is close to 100%, but the actual write speed is only 28MB/s, far below its peak speed (about 150MB/s). Furthermore, the write-intensive process will also interfere with the user queries that need to fetch the related data for the storage devices. Hence, disk I/Os can easily become the primary performance bottleneck of the database, especially for edge nodes with limited storage resources, and significantly affect the insert and query performances.

Due to high redundancy in time-series data collected from high-sampling-frequency sensors [1], [29], we can tolerate the loss of small amounts of data. Even in the presence of node failures, the lost data might be re-transferred from the sensors after recovery [26]. Therefore, we can relax the persistence guarantee to defer and merge many writes using TMTree to address the storing problem.

Time-series data tuples come into the system in chronological order, the tablets of same stream are generated at different points in time, and have different time ranges. It is thus hard to merge them to flush together. Therefore, EdgeDB presents **Time Merged K-ary Tree (TMTree)** to merge multiple tablets of different stream groups, which are collected by all sensors under the same time range, into a tablet group. These tablets are referred to as STR-tablets (**S**ame **T**ime **R**ange tablets), and may be generated at similar points in time. A tablet group can be flushed to disks with a single write operation, reducing the number of I/O operations. In addition, by storing these inter-tablet STR-tablets on disks contiguously, EdgeDB can gain higher inter-tablet join-query performance.

The TMTree structure is illustrated in Figure 4. The internal node in TMTree consists of two parts: a time window, which holds the time range of its subtree; and a pointer array, which records the addresses of its K child nodes. The leaf node is used to locate these STR-tablets. The leaf node consists of three parts: a time range, which is equal to the tablet window of these indexed tablets; a bitmap, which is n bits wide and indicates which sensors' tablets have been indexed by the leaf node, where n is the number of sensors processed by TMTree; and a pointer array, which has n pointers and records the memory addresses of these indexed tablets.

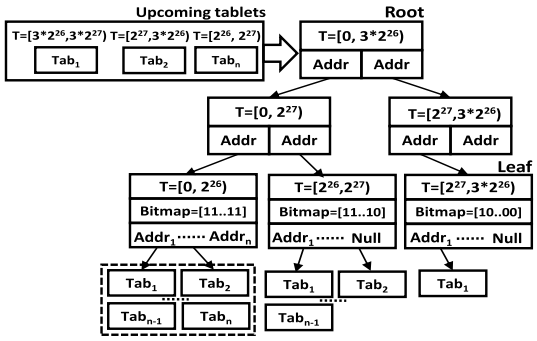


FIGURE 4. An example of a Time Merged K-ary Tree ($k = 2$) (Binary TMTree) with three layers for all sensors. The leaf nodes index STR-tablets. The tablets in dashed boxes can be written immediately.

Since each sensor has a unique id, when one tablet of a sensor is inserted into a leaf node, the system can determine a unique position in the bitmap (or in the pointer array) of the leaf node by hashing the sensor's id, and then updates the value at the position of the bitmap (or the pointer array).

Once a tablet is inserted into a $TPTree_d$, EdgeDB will launch the insert function of TMTree. The system first determines the leaf node to index the tablet according to its tablet window, and records the memory address of the tablet into the pointer array by hashing its id, then updates the bitmap of the leaf node to determine whether the tablet in this position already exists. Note that all STR-tablets are indexed by the same leaf node. EdgeDB periodically checks the bitmap in the leftmost leaf node, determines whether the leaf node has indexed all sensors' STR-tablets. If so, the system will merge these tablets together to flush to disks with a write operation. When the write operation is completed, EdgeDB reclaims the corresponding leaf node, and updates these tablets' on-disk addresses in the bottom index nodes of $TPTree_d$s which directly index these tablets. As shown in Figure 4, we find that the leftmost leaf node is full after the check process, these indexed tablets can be written to disks. The system frees this leaf node after the write operation, and inserts incoming tablets.

However, when some tablets are lost due to either sensor or network failures, the system will find that the corresponding leaf node does not index these tablets when checking its bitmap, and will wait for these lost tablets perpetually. The indexed STR-tablets will be held in memory and cannot be persisted. In order to avoid this, we set a timer for every leaf node. When a new tablet is inserted into a leaf node, the timer of the leaf node will be reset and start counting from 0. In the check process, EdgeDB also determines whether the timer of the leftmost leaf node exceeds the pre-defined time threshold. If so, the tablets without being indexed will be regarded as lost tablets. EdgeDB executes the write operation only for the existing tablets. If EdgeDB receives the lost tablets after their corresponding leaf nodes are reclaimed, the system keeps these tablets in memory, and flushes them to disks periodically.

We can reclaim these TMTree nodes in a timely manner after its indexed tablets have been completely flushed into storage devices. As a result, TMTree does not consume too much memory space. With TMTree, we not only reduce the number of I/O operations by deferring and aggregating small writes, but also potentially improve the inter-tablet join query performance, since the STR-tablets are stored sequentially on disks. When launching an inter-tablet query with a given time range, the system splits this time range into **n** tablet windows. And for each tablet window, EdgeDB only needs one read operation to fetch all data tuples produced by all sensors within the corresponding window. Afterwards, the system extracts desired results and sends them to users. Therefore, EdgeDB can reduce the number of read operations greatly when processing inter-tablet join query, and can potentially gain a higher inter-tablet join query performance.

IV. EVALUATION

In this section, we evaluate the efficacy of EdgeDB by comparing it against two time-series databases: BTrDB, a leading open-source time-series database system; and InfluxDB, the most popular time-series database according to DB-Engines Ranking [7]. The main evaluation metrics include insert, write and query performances, memory overhead and insert speed of TMTree.

We implement EdgeDB based on BTrDB using 2652 lines of go code. According to our design principles, we rewrite the index structure and query processing flow, and add the TMTree module to optimize the write procedure. The parameter K for the TPTree and TMTree structures is set at 64 in EdgeDB.

All experiments run on a testing machine with an Intel Xeon CPU E5-2650 2.00GHz processor and 32GB of memory. The testing machine is an off-the-shelf low-end server with the peak power of 150W, which is considered acceptable for a typical edge node in terms of processing capacity and power consumption [17]. The operating system is 64-bit Ubuntu 14.04, and the file system is ext4. The storage subsystem in our server contains a RAID-0 array containing 5 Western Digital 4TB HDDs, and a Samsung 250GB 750 EVO SSD. The peak write bandwidths are about 768MB/s for RAID, and 500MB/s for SSD, respectively, using the Linux dd command.

We use a real-world dataset acquired from a Gas sensor array under dynamic gas mixtures during 12 hours [8], [30]. The dataset has 4,178,504 data tuples, and each tuple includes a timestamp and 18 sampling values, which means that the dataset can be transformed into 18 time-series data streams, each of which contains 4,178,504 16 byte time-value pairs (8-byte timestamp, 8-byte value). We can further generate a large number of synthesized data streams and pairs with the dataset by adding small and random deviations into the raw sampling values. In addition, we also add the location information for each stream in order to merge correlated streams with the property.

A. INSERT PERFORMANCE

In this set of experiments, we evaluate the insert performance of EdgeDB. There are 7 types of inserts, i.e., InfluxDB, BTrDB, and 5 EdgeDB types. Each EdgeDB insert type corresponds to a different number of streams processed in batch (i.e., 1, 2, 4, 8, and 16) by classifying with location property. For BTrDB and InfluxDB, they process time-series data based on the *single-stream* approach. Every stream sends millions of sampling points (i.e., time-value pairs) in total to these systems, 4,096 points at a time. We flush all data to disks every five seconds.

Figure 5 shows the insert performance with various types of inserts as the number of streams increases. The insert performance of InfluxDB is much lower than those of the other two systems, and its highest performance is only 712k points per second. This is mainly caused by its logging mechanism and write amplification of LSM-tree-like structure. The logging mechanism is important to support recovery for key-value store. However, the high data redundancy in time-series data collected from high-sampling-frequency sensors makes the loss of small amounts of data tolerable [1], [29]. Therefore, such logging mechanism might be unnecessary for time-series data stores. For BTrDB and EdgeDB, when the number of streams is small, the processing capability of the machine cannot be fully utilized, and the insert throughput is low. When the number of streams is greater than 64, both systems start to deliver insert performances that are approaching their peaks. Note that when the number of the merged streams in EdgeDB is set to 1, its insert throughput is lower than BTrDB, because EdgeDB performs more processing steps, such as inserting tablets into TMTree, that are not necessary for only a merged stream. EdgeDB outperforms BTrDB when EdgeDB merges more than 4 streams to process together. In particular, when EdgeDB processes 16 streams in batches *EdgeDB[16]*, EdgeDB achieves up to 2.2× speedup over BTrDB when using RAID. This is due to the multi-stream merging mechanism that processes multiple data streams in batches. TMTree can defer and merge many small writes into a large sequential write, resulting in higher data processing speed and higher write performance. In addition, when using SSD, we can still achieve similar performance improvement.

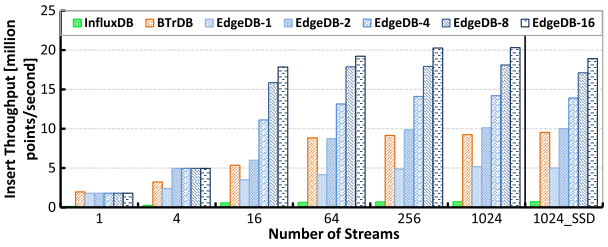


FIGURE 5. Insert throughput as a function of the number of streams. **BTrDB** and **InfluxDB** process all data streams based on the ***single-stream*** approach; **EdgeDB[*x*]**: every tablet organizes ***x*** data streams to be processed in batch (***x***: 1, 2, 4, 8, 16). The final set of experiments use SSD as the storage device, and others use RAID.

But the highest performance of EdgeDB on SSD is slightly lower than that on RAID, this is because SSD has a lower write performance compared to RAID.

B. WRITE PERFORMANCE

In order to evaluate the effectiveness of the TMTree technique, we conduct a set of experiments to measure and compare the write throughput and the number of write operations of EdgeDB and BTrDB when writing different amounts of data.

To measure the write throughput, we change the number of streams from 24 to 768, and every stream sends millions of data points to both systems. EdgeBD merges 12 streams together to process in batches. We flush all data to disks every five seconds. Figure 6 shows the maximum write performance that BTrDB and EdgeDB can achieve when processing different numbers of data points with different storage devices. While EdgeDB consistently outperforms BTrDB, EdgeDB's advantage over the latter is much more significant when using RAID than SSD. The maximum write throughputs for EdgeDB and BTrDB are 721MB/s, and 531MB/s respectively. The reason for EdgeDB performing comparably to or only slightly better than BTrDB for SSD is because SSD provides higher IOPS for write and can process a large number of write operations per second. The write throughput of EdgeDB almost saturates the sequential-write throughput of the storage devices.

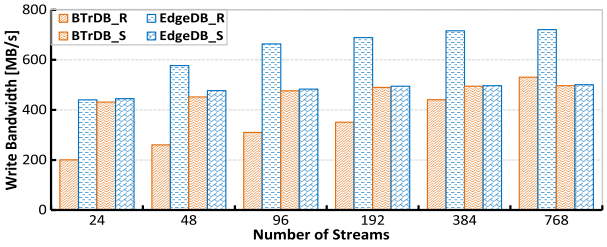


FIGURE 6. Write throughput as a function of the number of streams on RAID (*_R) or SSD (*_S), when periodically (i.e., 5s) flushing tablets into disks.

To understand the impact of flushing frequency on the performance, a higher frequency means that the amount of data lost is smaller when the node crashes or powers off. However, frequently flushing also degrades the writing performance due to the increasing number of small I/Os. We force writes to disks once tablets are ready to be written (i.e., once tablets are indexed for BTrDB, or once all tablets in leaf node of TMTree are available for EdgeDB). The write throughputs with high flushing frequency are shown in Figure 7, where EdgeDB is shown to have 3.6× higher write throughput than BTrDB, and the disk utilization of BTrDB is close to 100%, whereas EdgeDB only consumes 32.33% of disk bandwidth.

To measure the number of write operations issued by each system while processing a given number of streams, we vary the number of streams from 1 to 16, and each stream sends

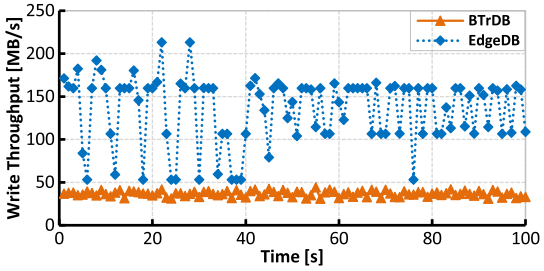


FIGURE 7. Write throughput at high flushing frequency, i.e., processing 1,000 streams. The write throughput is measured in two randomly selected stable 100-second windows, one for each system. The average write throughputs for EdgeDB and BTrDB are 135.48MB/s and 37.35MB/s, respectively.

the same number (1,000,000) of data points into both systems. Figure 8 reports the experimental results, which indicate that the number of write operations in BTrDB is directly proportional to the number of streams while this number remains constant in EdgeDB. Due to the lack of write optimization in the former, when finishing processing a tablet, BTrDB needs to write this tablet to hard disks, resulting in one write operation for each tablet and the number of write operations being proportional to the number of tablets and hence the number of streams. For EdgeDB, the number of tablets is independent of the number of streams, but the size of each write operation is proportional to the number of streams, due to its multi-stream merging mechanism.

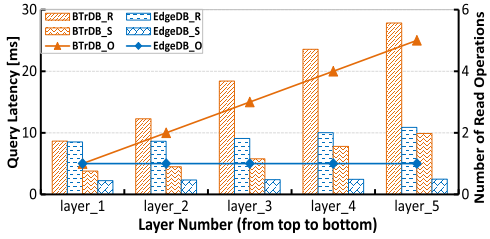


FIGURE 8. The number of write operations as a function of the number of streams.

Hence, by using TMTree, EdgeDB defers and batches many small writes into a large sequential write, as a result, EdgeDB can reduce the number of write operations, and improve the disk utilization, and make full use of write bandwidth of the underlying devices.

C. QUERY PERFORMANCE

For time-series databases, there are two important types of queries: one for aggregated values of a stream within a specified time range, referred to as **aggregation query**, and the other for the raw tuples of one or more streams within a time range, referred to as **join query**. In this subsection, we evaluate the performances of aggregation query and join query of EdgeDB. For each query, the index data and original data are only stored on disks, whereas the **TPTree_attr**s of all TPTrees are kept in memory.

1) AGGREGATION QUERY

Since the aggregated values of the tablets are kept in internal nodes of TPTrees, we use the query latency of internal node to measure the aggregation query latency.

Figure 9 shows the query latency and the number of read operations when querying the on-disk internal nodes of TPTrees in EdgeDB and BTrDB respectively. When querying lower-layer nodes, the latency of layer-by-layer query in BTrDB becomes higher, and the number of read operations also increases. On the other hand, EdgeDB uses TPEI and optimizes the disk layout of the index data, which enables the file offset of queried nodes to be easily calculated to fetch the requested data with a single I/O operation. For HDD, the latency of an aggregation query is close to a disk access latency (about 8ms), and up to 63% lower than the latency in BTrDB. Since SSD has extremely low access latency (about 100μs), the latency mainly depends on the time of accessing, migrating and parsing the nodes. In this case, EdgeDB only needs to process a node when querying, whereas BTrDB must process a number of nodes increasing with depth of the tree.

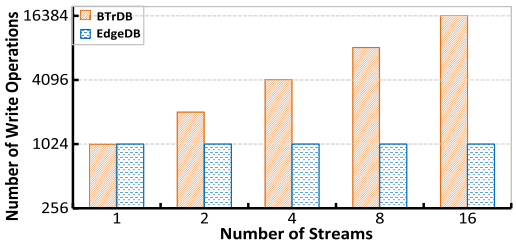


FIGURE 9. Latency results (bars) and the number of read operations (lines) of aggregation query when accessing the internal nodes in different layers of TPTrees with RAID (*_R) or SSD (*_S) as the underlying storage device.

In addition, for InfluxDB, the execution time of the aggregation query is nearly proportional to the length of the queried time range, and it takes up to 54ms to obtain the maximal value of 1024 points (which is the size of a shard in EdgeDB) of a stream with a continuous time range. Therefore, InfluxDB significantly underperforms EdgeDB, and even BTrDB.

When the queried time range is not aligned with the time window of any internal node, meaning that EdgeDB needs to fetch the leaf nodes containing raw points to compute the aggregated results, EdgeDB still gain performance improvement by avoiding fetching unnecessary on-disk internal nodes. More generally, the proposed index TPEI can reduce the response time for all time-range queries due to the optimized query procedure.

Furthermore, the memory usage of TPEI is at most 53MB for a stream as shown in Figure 10, much less that the space required to keep all index data (e.g., 20GB for a year's data) in memory, EdgeDB can reduce the memory usage greatly without incurring a high latency penalty.

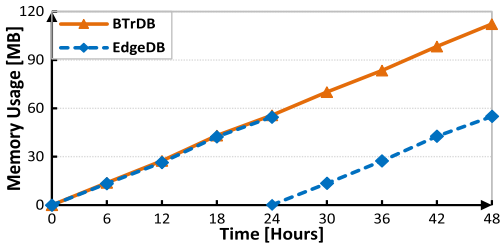


FIGURE 10. Memory usage of two index structures (TPEI in EdgeDB, TPTree in BTrDB) for a stream during two days.

2) JOIN QUERY

The join queries in this paper contain intra-tablet join queries and inter-tablet join queries. We measure the performances of these two types of join queries. To evaluate the performance of intra-tablet join query, we insert 16 time-series streams into each system, and each stream contains millions of data points. BTrDB and InfluxDB follow the *single-stream* approach to organize data points from every stream into a table, then build an index for every table separately; whereas EdgeDB organizes all data streams together, and then builds an index for them. For an intra-tablet join query which queries multiple correlated streams' data within a given time range, BTrDB and InfluxDB need to query every stream separately. Even though they can use multi-threading to process the queries in parallel, the number of I/O operations is not reduced and these I/Os cannot be easily parallelized, especially for HDDs. On the other hand, EdgeDB can query and fetch the involved data points in batches, entailing a (much) smaller number of larger I/O operations than BTrDB and InfluxDB. We specify the queried time range to contain 65,536 points from each stream.

Figure 11 shows the latency results for intra-tablet join queries as a function of the number of streams when using RAID. InfluxDB is the slowest, due to its inefficient index structures. For BTrDB and EdgeDB, as the number of streams increases, the intra-tablet query latency of BTrDB increases more significantly. This is because BTrDB needs more I/O operations to read more data, whereas EdgeDB requires almost the same number of read operations to fetch these data. EdgeDB achieves up to 57.3% decrease in intra-tablet query latency compared with BTrDB. The EdgeDB_WL and BTrDB_L in Figure 12 show the intra-tablet query perfor-

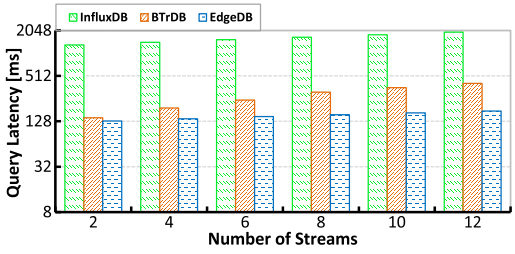


FIGURE 11. Latency of intra-tablet join query as a function of the number of streams using RAID.

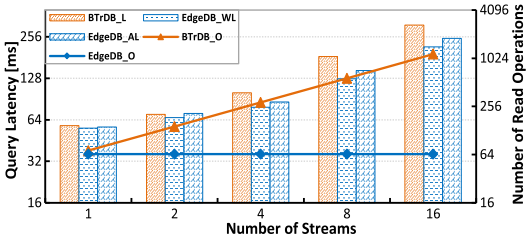


FIGURE 12. Latency results (bars) and the number (lines) of join query as a function of the number of streams using SSD. BTrDB_L: the latency of (intra-tablet or inter-tablet) join query; EdgeDB_WL(AL): the latency of intra-tablet (inter-tablet) query.

mances using SSD. EdgeDB can still achieve 43.1% performance improvement due to fewer I/O operations, even though SSD provides higher read performance.

We evaluate the performance of inter-tablet join query by querying data streams from different tablets within a given time range. To avoid effects of intra-tablet join query, in the following experiments, EdgeDB also processes each stream separately, but stores the streams within the same time ranges together. For BTrDB and InfluxDB, the query requires fetching the data collected by every stream separately, meaning that querying for **n** streams' data requires **n** simple query operations. However, EdgeDB needs only one read operation to fetch all data collected by all streams within a tablet window, since these data within the same tablet window are stored on disks sequentially by using the TMTree. We also specify the queried time range to contain 65,536 points from each stream. Figure 13 compares join query performances among the three systems with the different numbers of streams when using RAID as the storage device. The query performance of InfluxDB is again significantly lower than the other two systems. When querying data produced by a small number of streams, BTrDB has similar performance to EdgeDB, due to their similar query procedure under this circumstance. However, as the number of queried streams grows, EdgeDB's well-designed data layout on disks allows it to provide a better join query performance than BTrDB, achieving up to 55.9% reduction in query response time.

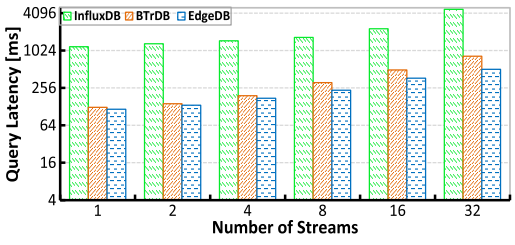


FIGURE 13. Latency results of inter-tablet join query as a function of the number of streams using RAID.

For SSD, EdgeDB can still achieve small performance improvement as presented in Figure 12. In addition, when querying in EdgeDB, there is no significant change in the number of read operations, regardless of the number of streams. While in the BTrDB system, the number of read operations is proportional to the number of streams. In summary, the main cause of performance degradation in EdgeDB, as the number of streams increases, is because it has to read and process more data for each read operation. In addition, BTrDB also induces a large number of discontinuous I/Os as it fetches requested data from different files, resulting in a poor query performance, especially for HDD.

D. MEMORY USAGE AND INSERT SPEED OF TMTREE

TMTree is an in-memory and auxiliary index structure designed to merge tablets together in order to reduce I/O operations. As such, TMTree should not consume substantial memory resource, while maintaining high insert performance.

To measure the memory usage of TMTree, we spawn 1,000 threads to simulate 1,000 sensors to send tablets into TMTree. Since different sensors have different network conditions, we set four different data transmission delays for all sensors as listed in the figure title of Figure 14. Every thread sends a new tablet each 4 seconds. And before sending a tablet, every thread will sleep for its pre-defined transmission delay. Upon receiving the tablet, TMTree inserts it into the leaf node.

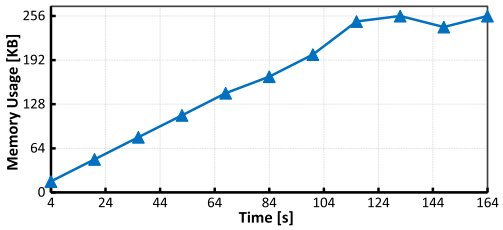


FIGURE 14. Memory usage of TMTree when inserting tablets. We setup four different network conditions for all sensors, including a less-than 5s transmission delay for 90% sensors, a less-than 30s delay for 5% sensors, a less-than 60s delay for 3% sensors, a less-than 120s delay for 2% sensors.

Figure 14 shows the number of TMTree nodes in memory when using TMTree to index these tablets. As tablets are continuously inserted into TMTree, EdgeDB needs to create the internal nodes and leaf nodes to index these tablets, and cannot reclaim any nodes due to not indexing enough tablets. After about 120 seconds, the first leaf node is full, and these indexed tablets can be written to disks, and the leaf node can be reclaimed. Since then, we can reclaim old nodes in a timely manner, and create new nodes for incoming tablets. As a result, the total number of nodes remains steady, about 30 (at most 8KB for a node). Thus, EdgeDB just needs to allocate 240KB memory space for each TMTree.

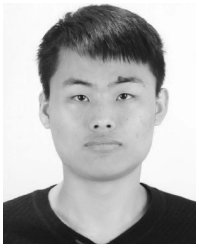
In addition, we measure the insert performance of TMTree. The average insert speed is 53 thousand tablets per second, and it is enough for EdgeDB with hundreds or thousands of data points in each tablet.

V. CONCLUSION

We propose an efficient time-series database EdgeDB for edge nodes with three novel mechanisms. First, the multi-stream merging mechanism is proposed to organize multiple correlated streams, which can be queried together and processed in batches. Second, EdgeDB adopts a memory-efficient index, TPEI, to improve the time-range query performance by optimizing the disk layout of index. Finally, EdgeDB maximizes write throughput of disks by using TMTree to reduce small writes. The extensive experiments driven by real sensor datasets show that EdgeDB achieves higher write throughput, higher query performance, better insert throughput, and lower resource requirement than existing time-series databases.

REFERENCES

[1] N. Agrawal and A. Vulimiri, "Low-latency analytics on colossal data streams with summarystore," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 647–664.
[2] M. P. Andersen and D. E. Culler, "BTrDB: Optimizing storage system design for timeseries processing," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 39–52.
[3] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A distributed multi-resolution time-series datastore," in *Proc. IEEE 34th Real-Time Syst. Symp. (RTSS)*, Dec. 2013, pp. 288–297.
[4] Cassandra. (Jan. 2018). *Apache Cassandra*. [Online]. Available: http://cassandra.apache.org/
[5] H. Chihoub and C. Collet, "A scalability comparison study of data management approaches for smart metering systems," in *Proc. 45th Int. Conf. Parallel Process. (ICPP)*, Aug. 2016, pp. 474–483.
[6] Datafloq. (2018). *Self-Driving Cars Will Create 2 Petabytes of Data*. [Online]. Available: https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172
[7] *DB-Engines*. (Jan. 2018). [Online]. Available: https://db-engines.com/en/ranking/time+series+dbms
[8] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring," *Sens. Actuators B, Chem.*, vol. 215, pp. 618–629, Aug. 2015.
[9] M. Gabel, A. Schuster, and D. Keren, "Communication-efficient distributed variance monitoring and outlier detection for multivariate time series," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 37–47.
[10] A. Giachanou and F. Crestani, "Tracking sentiment by time series analysis," in *Proc. 39th Int. ACM SIGIR Conf. Res. Develop. Inf. Retr. (SIGIR)*, 2016, pp. 1037–1040.
[11] D. Hong, H. Wang, and K. Whitehouse, "Clustering-based active learning on sensor type classification in buildings," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2015, pp. 363–372.
[12] *Influxdata*. (Jan. 2018). [Online]. Available: https://www.influxdata.com/time-series-platform/
[13] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "ModelarDB: Modular model-based time series management with spark and cassandra," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1688–1701, 2018.
[14] A. Jha, S. Ray, B. Seaman, and I. S. Dhillon, "Clustering to forecast sparse time-series data," in *Proc. IEEE 31st Int. Conf. Data Eng. (ICDE)*, Apr. 2015, pp. 1388–1399.
[15] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3229–3242, Nov. 2017.
[16] G. Luo, K. Yi, S.-W. Cheng, Z. Li, W. Fan, C. He, and Y. Mu, "Piecewise linear approximation of streaming time series data with max-error guarantees," in *Proc. IEEE 31st Int. Conf. Data Eng. (ICDE)*, Seoul, South Korea, Apr. 2015, pp. 173–184.
[17] X. Lyu, H. Tian, and L. Jiang, "Selective offloading in mobile edge computing for the green Internet of Things," *IEEE Netw.*, vol. 32, no. 1, pp. 54–60, Jan./Feb. 2018.
[18] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Carlsbad, CA, USA, Oct. 2018, pp. 477–489.
[19] F. Mei, Q. Cao, H. Jiang, and L. T. Tintri, "LSM-tree managed storage for large-scale key-value store," in *Proc. Symp. Cloud Comput. (SoCC)*, Santa Clara, CA, USA, Sep. 2017, pp. 142–156.
[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inform.*, vol. 33, no. 4, pp. 351–385, 1996.
[21] ORACLE. (Jan. 2018). *MySQL Home Page*. [Online]. Available: https://www.mysql.com/
[22] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.
[23] J. Qi, R. Zhang, K. Ramamohanarao, H. Wang, Z. Wen, and D. Wu, "Indexable online time series segmentation with error bound guarantee," *World Wide Web*, vol. 18, no. 2, pp. 359–401, 2015.
[24] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankovskii, "Solving big data challenges for enterprise application performance management," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1724–1735, 2012.
[25] REDIS. (Jan. 2018). *Redis Home Page*. [Online]. Available: http://redis.io/
[26] S. Rhea, E. Wang, E. Wong, E. Atkins, and N. Storer, "LittleTable: A time-series database and its uses," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 2017, pp. 125–138.
[27] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, Jun. 2013.
[28] S. B. Taieb, J. Yu, M. N. Barreto, and R. Rajagopal, "Regularization in hierarchical time series forecasting with application to electricity smart meter data," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 4474–4480.
[29] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "ADMin: Adaptive monitoring dissemination for the Internet of Things," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.
[30] (Jan. 2018). *UCI. Uc Irvine Machine Learning Repository*. [Online]. Available: http://archive.ics.uci.edu/ml/datasets
[31] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. N. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, "FarmBeats: An IoT platform for data-driven agriculture," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 515–529.
[32] L. Wang, R. Cai, T. Z. J. Fu, J. He, Z. Lu, M. Winslett, and Z. Zhang, "Waterwheel: Realtime indexing and temporal range query processing over massive data streams," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Paris, France, Apr. 2018, pp. 269–280.
[33] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Trans. Commun.*, vol. 64, no. 10, pp. 4268–4282, Oct. 2016.
[34] L. Wei and J. M. Mellor-Crummey, "Automated analysis of time series data to understand parallel program behaviors," in *Proc. 32nd Int. Conf. Supercomput. (ICS)*, Beijing, China, Jun. 2018, pp. 240–251.
[35] Y. Zhao, C. Qian, L. Gong, Z. Li, and Y. Liu, "LMDD: Light-weight magnetic-based door detection with your smartphone," in *Proc. 44th Int. Conf. Parallel Process. (ICPP)*, Sep. 2015, pp. 919–928.



YANG YANG received the B.S. degree in software engineering from the Nanjing University of Aeronautics and Astronautics, in 2015. He is currently pursuing the Ph.D. degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. He was involved in optimizing time-series databases. His research interests include the file system and NVM storage.

QIANG CAO received the B.S. degree in applied physics from Nanjing University, in 1997, and the M.S. degree in computer technology and the Ph.D. degree in computer architecture from the Huazhong University of Science and Technology, in 2000 and 2003, respectively, where he is currently a Full Professor with the Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture, large-scale storage systems, and performance evaluation. He is also a Senior Member of the China Computer Federation (CCF) and ACM.

HONG JIANG received the B.Sc. degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the M.A.Sc. degree in computer engineering from the University of Toronto, Toronto, ON, Canada, in 1987, and the Ph.D. degree in computer science from Texas A&M University, College Station, TX, USA, in 1991. He has been with the University of Nebraska–Lincoln, since 1991, where he was a Willa Cather Professor of computer science and engineering. He served as a Program Director with the National Science Foundation, from January 2013 to August 2015. He has graduated 16 Ph.D. students who upon their graduations either landed academic tenure-track positions in the Ph.D.-granting U.S. institutions or were employed by major U.S. IT corporations. He is currently the Chair and a Wendell H. Nedderman Endowed Professor with the Computer Science and Engineering Department, University of Texas at Arlington. He has more than 300 publications in major journals and international conferences in these areas, including the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON COMPUTERS, the PROCEEDINGS OF THE IEEE, the *ACM Transactions on Architecture and Code Optimization*, the *ACM Transactions on Storage*, the *Journal of Parallel and Distributed Computing*, ISCA, MICRO, USENIX ATC, FAST, EUROSYS, LISA, SIGMETRICS, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, and ICPP. His current research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, and performance evaluation. His research has been supported by NSF, DOD, the State of Texas and the State of Nebraska, and industry. He is a member of the ACM.