PISA: an Index for Aggregating Big Time Series Data

Xiangdong Huang[†], Jianmin Wang[†♯], Raymond K. Wong[§], Jinrui Zhang[†], Chen Wang[†]
[†]School of Software, Tsinghua University, [♯]TNList, Institute for Data Science, Beijing 100084, China
[§]School of Computer Sc. & Eng., University of New South Wales, Sydney, Australia
huangxd12@mails.tsinghua.edu.cn, jimwang@tsinghua.edu.cn,
wong@cse.unsw.edu.au

ABSTRACT

Aggregation operation plays an important role in time series database management. As the amount of data increases, current solutions such as summary table and MapReduce-based methods struggle to respond to such queries with low latency. Other approaches such as segment tree based methods have a poor insertion performance when the data size exceeds the available memory. This paper proposes a new segment tree based index called PISA, which has fast insertion performance and low latency for aggregation queries. PISA uses a forest to overcome the performance disadvantages of insertions in traditional segment trees. By defining two kinds of tags, namely code number and serial number, we propose an algorithm to accelerate queries by avoiding reading unnecessary data on disk. The index is stored on disk and only takes a few hundred bytes of memory for billions of data points. PISA can be easily implemented on both traditional databases and NoSQL systems, examples including MySQL and Cassandra. It handles aggregation queries within milliseconds on a commodity server for a time range that may contain tens of billions of data points.

CCS Concepts

•**Information systems** → **Summarization;** *Spatial-temporal systems; Search engine indexing;*

Keywords

temporal data; aggregation index

1. INTRODUCTION

In recent years, more and more sensor data is collected for monitoring, analysis and forecasting. This data is usually organized along the time dimension to form a huge amount of time series data. For example, there are more than 100,000 meteorological ground stations in China. They generate about 100 billion, high dimensional data points per year. Therefore, an efficient database system which can handle such large amounts of time series data is needed.

Table scan is inefficient for time series data, as valuable information is submerged in the data point ocean. On the other hand, aggregation queries give users more valuable information for the purposes of monitoring, analysis and forecasting. For example, consider a query "`select max(value) from table where time > 2016.4.1 8:00 and time < 2016.4.1 9:00`". If the query result is 10 while the range of the value should be within $(2, 8)$, we know that something unusual (i.e., anomaly) happened within that period. In practice, start time and end time in the above query can be arbitrary: e.g., one week, one year, or just from 8:01 to 8:07 on 2016.4.1. Users expects a low latency for these queries regardless of the data size.

Aggregation query has previously been studied in various flavors [5]. Many of these methods do not consider the Big Data scenarios, in which each time series has billions of data points. Some relational databases and NoSQL systems currently use summary tables (aka synopsis table) or MapReduce based methods to address the aggregation performance requirements [19, 7, 9]. Due to the increasing amount of data points, the response time can vary from several seconds to several minutes [17, 16]. However, in many analytic and forecasting applications such as weather forecasting, users demand to get aggregation results within milliseconds and ideally on commodity computer hardware.

Tree-based indexes, such as those based on segment tree [23, 13], have been proposed to accelerate aggregation queries. Segment tree [3] is a static structure [13] suitable for aggregation operations but has some weaknesses: (1) Constructing a segment tree requires all the timestamps of each data point to be known a priori [12], which is impossible for a time series. (2) When storing a segment tree on disk, many unnecessary internal nodes will be read from the disk because the query algorithm requires traversing the tree from the root to leaf nodes. Besides, storing the tree on disk is inevitable because the size of the tree will likely exceed the size of available memory. Some works, such as SB-tree [23] and Balanced tree [13], have been done for solving the first problem but none has addressed the second weakness. Furthermore, if the tree is stored on disk rather than in memory, for each insertion operation, many internal nodes need to be updated (i.e., too many IO operations). As a result, the insertion performance of the above proposed methods will degrade rapidly when the data size increases.

In this paper, we propose PISA, a Persistent Index for Segmented Aggregation, to optimize aggregation queries on

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983775

an arbitrary time range for a large-scale time series. PISA is based on the segment tree structure but has differences: for an internal node in the tree, we do not store the earliest and latest timestamp of the time range that the node can cover. Instead, we define two tags, *code number* and *serial number*, and store them on each internal and leaf node.

PISA overcomes the weaknesses of a traditional segment tree through the following: (1) It is a forest instead of a single tree. Meanwhile, we propose the insertion algorithm to combine two trees at the appropriate time; (2) It gives up traversing the tree and reading internal nodes level by level. By using the tags in nodes, we can avoid reading unnecessary nodes and only access nodes which are part of the result. This is important because PISA is stored on disk rather than in memory.

The construction and query processes are as follows. Firstly, when the database receives a fixed number of data points (e.g. 100 points), PISA pre-calculates their aggregation values and then updates the index accordingly. When an internal node or leaf is constructed, PISA stores two tags, *code* and *serial number* in the node. Secondly, given a time range, PISA splits the time range into sub-ranges and then finds the related tree nodes according to the tags. In this way, PISA can get the aggregation value of tens of billions of data points in milliseconds on a commodity server. In PISA, it only takes several hundred bytes of memory for a very large amount of time series data. Besides, PISA can be implemented easily on existing NoSQL systems and relational databases such as Cassandra and MySQL.

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

- We extend the segment tree structure as a segment forest to supporting fast insertion operations.
- In each node, we store two kinds of tags, *code number* and *serial number*, rather than the earliest and latest timestamps of the time range that the node can cover. A new query algorithm based on the tags helps us to avoid reading unnecessary nodes on disk.

The paper is organized as follows: Section 2 presents the related works and Section 3 defines the query requirements formally. Sections 4 and 5 introduce how to build the index and how to query by the index. In Section 6, we implement the index on Cassandra and MySQL. We then compare the index with 5 other methods on Cassandra and MySQL built-in indexes. Finally, Section 7 concludes the paper.

2. RELATED WORKS

This section summarizes the following works that are related to PISA: (1) synopsis table and its improvements in traditional databases; (2) Map-reduce based aggregations in NoSQL systems; (3) tree-based aggregation indexes; and (4) others.

Synopsis tables [20], also known as summary tables [8], are a common way to accelerate the processing of aggregation queries. With synopsis table, a database system calculates the aggregation value periodically to generate a coarse granularity table. When the database receives an aggregation request, it calculates the final result by scanning the synopsis table rather than reading the original data [20]. Though the size of synopsis table is smaller than the original data, there can still be too many records. For example, when the original data table has billions of records, its synopsis table may have millions of records. This is still too slow for a big database.

The query response time of synopsis tables can be accelerated by sacrificing the accuracy of results, as shown in PTA [6, 11]. PTA allows users to define the maximum size of the synopsis table (or the maximum error bound of the aggregation result). Then PTA uses a dynamic program to reduce the table size. In this way, PTA can reduce the synopsis table size and hence the query processing time significantly. However, users may lose too much of the accuracy to achieve the required performance when data size is large.

Only a few NoSQL database systems support aggregation functions, as surveyed by [15]. Many of them use MapReduce [4] to handle efficient aggregation operations [9, 7, 18]. For each aggregation operation, the query engine filters the data in the "map" stage and calculates the final result in the "reduce" stage. For example, in Impala [7], aggregations are executed as local pre-aggregations following by a merge operation. Because MapReduce needs a lot of IO operations, many works focused on how to optimize MapReduce to accelerate the aggregation operations [18, 9, 10]. In addition to MapReduce, some other technologies were also proposed for processing aggregations in NoSQL databases. For example, MongoDB [1] uses aggregation pipeline that combines the idea of MapReduce and Linux pipelines.

Tree-based indexes for the aggregation query were surveyed in [5] and many of them were based on segment trees. The definition of a segment tree can be found in [12]. To build a segment tree, users need to know all the data points first. Then a full binary tree is built. Each node in the tree covers a data point or a time range. Suppose node i has two children j and k, which cover $[t_{j_s}, t_{j_e}]$ and $[t_{k_s}, t_{k_e}]$ respectively. Then node i will cover the time range $[t_{j_s}, t_{k_e}]$. Node i stores two timestamps, t_{j_s} and t_{k_e}, and the aggregation result of the time range. Given an aggregation query, we need to traverse the tree from the root to leaves. We read the two timestamps in each node to determine which child of the node should be traversed next. Therefore, for big data, we may need to read many internal nodes in order to get to the leaf nodes that contribute to the final aggregation result. As a result, relatively high overhead cost is spent on reading these nodes. This cost can be very significant when the tree is stored on disk.

By combining segment tree and other tree structures, variants of segment tree were proposed. For example, SB-tree [23] transforms segment tree as a b-tree so that the index supports insertion and the depth of the tree is reduced. In another example, Balanced-tree [13] uses AVL or red-black tree to implement segment tree in order to support insertion. However, the actual insertion operation is complex. It includes split or rotation operation as well as update operation on internal nodes. Hence insertions on SB-tree and Balanced-tree are not efficient especially when the data is large or the trees are stored on disk.

Finally, sliding window aggregations were also drawn significant attention from the community. It was first introduced in [14] and new development and research had been ongoing (e.g., [2, 21]). Different from the focus of this paper, sliding window aggregation only aggregates the latest data with a sliding manner [21], e.g., "the average value for the data from the last 5 minutes while sliding the time window per 1 minute". Therefore, indexes for answering such queries can drop stale data and only need to cache a small

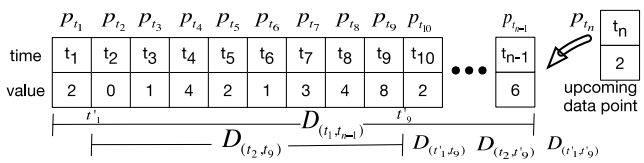

Figure 1: Time series data

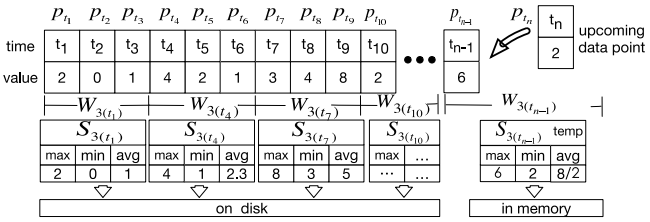

Figure 2: Window summary

amount of data points in memory. Comparing with sliding window aggregation, queries that PISA focuses on have more relaxed query conditions and hence needs to deal with larger amounts of data. Furthermore, sliding window aggregation can also be handled by segment tree (e.g., via in-memory segment tree [21]).

3. PROBLEM SPECIFICATION

In this section, we define the data to be processed and query requirements.

Data point p, the basic data item, can be described as a pair $p =< t, v >$, which means that at time t, the database receives a data point and the value is v. We use T to represent the timestamp set and p_t to represent the point in which $t \in T$. A **time series** $D_{(s,e)}$ represents a time series segment which consists of all data points between time s and e, while these points are sorted by their timestamps. Formally, $D_{(s,e)} = \{p_t | t \in T \wedge t \geq s \wedge t \leq e\}$. In a real-time database, the time series D increases constantly by receiving new points in a fixed or irregular frequency.

Figure 1 illustrates an example of time series data. $D_{(t_1,t_{n-1})}$, which consists of data points $p_{t_1} \sim p_{t_{n-1}}$, is the longest time series in this case. Data point p_{t_n} will arrive in the future. $D_{(t_2,t_9)}$ is a segment time series of $D_{(t_1,t_{n-1})}$. Given a timestamp t'_1 and $t_1 < t'_1 \leq t_2$, $D_{(t'_1,t_9)}$ is equal to $D_{(t_2,t_9)}$, because there is no data point between (t'_1, t_2). $D_{(t_2,t'_9)} = D_{(t'_1,t'_9)} = D_{(t_2,t_9)}$ as well, if $t_9 \leq t'_9 < t_{10}$.

Operation $\mathbb{A}(D)$ represents applying **aggregation operation** $\mathbb{A}$ (e.g. max(), avg(), count() and so on) on the time series D.

$Q_{\mathbb{A}}(s, e)$ is a **query** for aggregation operation $\mathbb{A}$ on $D_{(s,e)}$. For example, $Q_{max}(t_2, t_9)$ represents getting the maximum value among $p_{t_2} \sim p_{t_9}$. Our goal is to answer the query like this in millseconds even though there are billions of data points and only a limited amount of memory is available (e.g., several KB).

4. BUILDING PISA

4.1 Window summary

We use **window** to split a time series and as the basic unit for calculating the aggregation value. A **window** $W_{k(s)}$ is a specific time series that has k points and p_s is the first data point in the window. Window can be formalized as $W_{k(s)} = \{p_s\} \cup \{p_t | \exists e \in T \wedge |D_{(s,e)}| = k \wedge p_t \in D_{(s,e)}\}$.

According to the definition of window, we can split a time series D as follows: the first k points form a window, then the next k points form another one. Suppose $D_{(s,e)}$ has $nk + m$ points, in which $m < k$. For the first nk points, there will be n windows to wrap them. In this way, all the windows form a partition of D (excluding the last m points). For the next m points, we wrap them in memory until the next $k - m$ points arrive and form a new window. For a given time series, we require all the windows to have the same size, i.e., same number of points.

For each window W, we apply $\mathbb{A}$ on W and generate a **window summary** information S. S stores the aggregation result of $\mathbb{A}(W)$ and will be persistent as a part of the PISA. Formally, $S = \mathbb{A}(W)$. If a window summary represents the aggregation value of k points, we name it k-sized window summary.

Figure 2 illustrates how to assign windows and calculate window summaries. In this case, each 3 points form a window. Therefore, we split the time series $D_{(t_1,t_{12})}$ into 4 windows: $W_{3(t_1)} \sim W_{3(t_{10})}$, and a series of window summaries $S_{3(t_1)} \sim S_{3(t_{10})}$ are generated. The aggregation operations include Max, Min and Avg. After these window summaries are generated, they are then stored on disk. For the last two points $p_{t_{n-1}}$ and p_{t_n}, we generate a window summary in memory, calculate the temporary aggregation value and wait for the upcoming points to form a new window. Finally, this new window summary will be stored on disk as well.

All the window summaries form a summary of the original time series. In PISA, each leaf node contains a window summary and some tags, which will be introduced in Section 4.2.1. Compared to putting a single data point into a leaf node, using one window summary to generate one leaf node can reduce the number of nodes in the index.

4.2 The index

The goals of the PISA are: (1)getting the aggregation value on any arbitrary time range, (2) inserting new data points in real-time, (3) avoiding reading unnecessary data from disk. To achieve the first goal, we use a tree-based structure similar to segment tree. For the second goal, we design a forest structure and an insertion algorithm. For the third goal, we give up the traditional search method, which need to read internal nodes level by level.

PISA: We maintain **Persistent Index for Segment Aggregation** as a set (or a forest) of complete binary trees, which are called as **persistent segment tree**s (PST). Formally, $PISA = \{Tree_i | i \in I\}$. I is the integer set and i is the code of the tree, which will be discussed in detail in Section 4.2.1.

4.2.1 *Persistent segment tree structure*

Persistent segment tree (PST) is a complete binary tree. The property of PST is similar with traditional segment tree, while the contents of leaf and internal nodes are different from that of a traditional segment tree.

To define the node in a PST, we define the aggregation of two window summaries at first. Suppose $S_{k(t_i)}$ and $S_{k(t_j)}$ represent the aggregation values of two time series segments $D_{(t_i,t_{k1})}$ and $D_{(t_j,t_{k2})}$ and there is no data point between

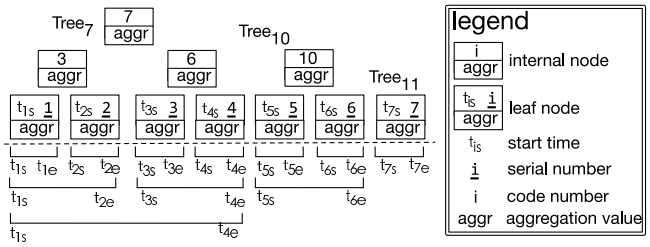

Figure 3: An example of PISA

the two time series segments, we can apply an aggregation operation on the two window summaries and then generate a new window summary $S_{2k(t_i)}$, which represents the aggregation value of time series segment $D_{(t_i,t_{k2})}$. Similarly, two $2k$-sized window summaries can be combined as a $4k$-sized window summary. Besides, two window summaries with different sizes can not be combined. Formally, $\mathbb{A}(S_{2^n k(t_i)}, S_{2^n k(t_j)}) = S_{2^{n+1} k(t_i)}$.

Leaf node (LF). The leaf node of the PST consists of 3 parts: a value, which is a k-sized window summary; a name, which is the start time of its window summary; and a tag, *serial number*. Leaf nodes in the index and window summaries in the whole time series are one-to-one mapping.

Internal node (IN). The internal node of the PST consists of 2 parts: a $2^n k$-sized window summary, which is the aggregation result of the window summaries in its children; and a tag, *code number*.

We define two types of tags in the tree, one is *serial number* and the other is *code number*.

Serial number (SN). Serial number starts with 1. Each leaf node owns a unique serial number. When we assign a serial number i to a leaf node, the serial number of the next leaf node is $i+1$. Because each leaf is mapping to a k-sized window summary, the maximal serial number is equal to the number of window summaries. Internal nodes have no serial number tag.

Code number (CN). Code number starts with 1 and will increase 1 when the index adds a new (leaf or internal) node. Nodes get code numbers according to the post order traversal of the tree. Internal nodes have code numbers.

A leaf node which stores $S_{k(t_i)}$ is $LF(t_i)$. An internal node whose code number is i is IN_i. Considering a leaf node whose serial number is j as a special internal node, then we can assign a code number, $cn(j)$, to the leaf. Then a leaf node whose code number is i $(= cn(j))$ can be marked as LF_i. The calculation of $cn(j)$ will be introduced in Section 4.2.3 (Equation 2). Besides, a tree can be named as the code number of its root node.

The top of Figure 3 illustrates a PISA with 3 PSTs, $PISA = \{Tree_7, Tree_{10}, Tree_{11}\}$. The leaf node stores its serial number, a k-sized window summary ('aggr' in the figure) and the start time of the window summary ($\mathtt{t}_{is}$ in the figure). The internal node stores its code number and the aggregation value that is calculated according to its children. However, the internal node does not need to store which nodes are its children or the locations of its children.

We use t_{is} and t_{ie} to describe the timestamps of the first point and the last point in window $W_{k(t_i)}$ (that is, $t_{is} = t_i$). The bottom of Figure 3 is an upside down mirror of PISA, while circles are replaced with segment lines. In Figure 3, a higher level in a tree holds longer time series than its children. For example, LF_{t_1} represents the aggregation value of $D_{(t_{1s},t_{1e})}$ and IN_7 represents the aggregation value of $D_{(t_{1s},t_{4e})}$.

4.2.2 Storage

It is expensive to store the whole forest in memory for just supporting aggregation operations. For a complete binary tree, the node number is $2 \times n - 1$ while n is the number of leaf nodes. The number of nodes in PISA is not greater than $2 \times n - 1$ (and will be equal to it when there are $n = 2^m$ leaf nodes). It is still impossible to store all the nodes in memory to handle tens of billions of data points. For example, if $k = 100$ for a window and we have 2^{34} data points in total. For each window summary, we suppose it only stores the maximal, minimal, average and count value in float format (8 bytes). Then we need $(2 \times 2^{34}/100 - 1) \times (8 \times 4) = 11GB$ memory. Therefore, we store all the nodes on disk and only hold root nodes of trees into memory (the reason will be discussed in Section 4.2.3). It is easy to prove that for n points and k-sized window summary, the maximum number of root nodes is $\lfloor log_2(n/k) \rfloor + 1$. For 2^{34} data points, the maximum number of root nodes is $\lfloor log_2(2^{34}/100) \rfloor + 1 = 28$. The memory cost is $28 \times (8 \times 4) = 896$ Byte.

More accurately, if there are m leaf nodes, the number of root nodes $f(m)$ is:

$$f(m) = \begin{cases} 1, & if \quad m == 2^t \wedge t \in I \\ 1 + f(m - 2^t), & if \quad 2^t < m < 2^{t+1} \wedge t \in I \end{cases} \tag{1}$$

Because PISA is a forest of complete binary trees, the number of total nodes is not greater than $2 \times m - 1$, in which m is the number of leaf nodes. How to store the nodes on disk is not invariable, but the store engine needs to support three operations:

(A0) Serialize a leaf or internal node on disk. We use operations $IO.write(LF)$ or $IO.write(IN)$ to represent it.

(A1) Given the start time of a leaf or the code number of an internal node, the store engine can get the node quickly. We mark the operation as $IO.get(t_i)$ or $IO.get(i)$. t_i is the start time of a leaf and $IO.get(t_i) = LF(t_i)$. i is a code number and $IO.get(i) = IN_i$. A batch read operation $IO.batch\_get(\{i, j...\})$ is optional.

(A2) Given a timestamp of a data point, the store engine can get the leaf node whose start time is greater or less than the given timestamp. We mark the operation as $IO.gt(t_i)$ or $IO.ls(t_i)$. The formal definition of the result for the operation will be described in Section 5.1. To archive the operation with low latency, we recommend to store all the leaves together and sort them according to their start timestamps.

Most databases can provide the above abilities. We will show how to use SQLs to implement PISA in Section 5.4.

4.2.3 Insertion

The insertion algorithm can be summarized as follows. When a k-sized window summary is generated, we add a leaf node into the forest and add tags into the node. The new leaf node forms a single node tree. If there are two trees which meet combining conditions below, we combine them into a new tree.

Conditions for combining. Mark the depth of $Tree_i$ as $depth(Tree_i)$, if two trees meet:

(1)$Tree_i, Tree_j \in PISA \wedge i < j$,

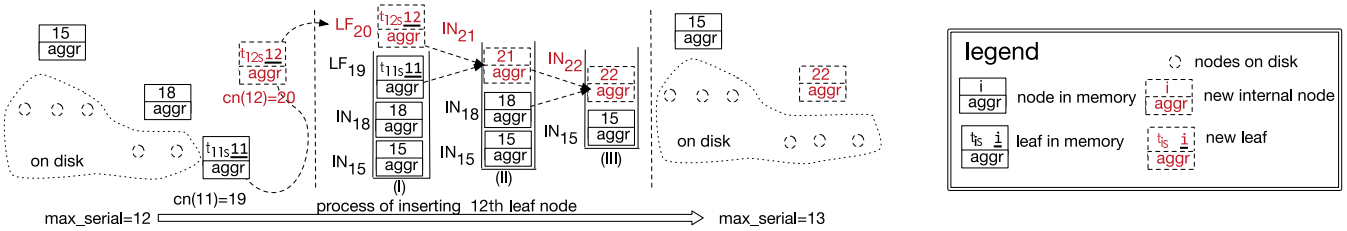

Figure 4: Leaf node insertion in PISA

(2) $\nexists Tree_m \in PISA \wedge i < m < j$,
(3) $depth(Tree_i) = depth(Tree_j)$.

We can merge $Tree_i$ and $Tree_j$ together by generating a new node whose left child is the root node of $Tree_i$ and the right child is the root node of $Tree_j$. The window summary of the new node is $\mathbb{A}(Tree_i.root\_node.S, Tree_j.root\_node.S)$.

Figure 4 illustrates the process of inserting the 12th leaf node into the index. Before adding $LF_{t_{12s}}$ (i.e., LF_{20}), $PISA = \{Tree_{15}, Tree_{18}, Tree_{19}\}$. After adding $LF_{t_{12s}}$, a new tree $Tree_{20}$ (a single node LF_{20}) is generated. $Tree_{19}$ and $Tree_{20}$ meet the conditions of combining, so they can be merged together (step (I) in the figure).

In the above example, 20 is calculated according to 12 (12th). The calculation method is as follow:

SN and CN calculations. If the index has handled $i-1$ leaf nodes and prepares to add the ith leaf node, then the serial number of the new leaf node is i. The code number is:

$$cn(i) = \begin{cases} 2 \times i - ones(i), & if \quad i\%2 == 1 \\ 2 \times (i-1) - ones(i-1) + 1, & otherwise \end{cases} \tag{2}$$

with ones(i) representing the hamming weight [22] of i, meaning the number of "1" in i with the binary format. Function $ones()$ can be implemented with many algorithms. The complexity of the best algorithm is $O(1)$ for a 64-bit long format number.

Given a serial number i, the largest code number $LF_{cn(i)}$ brings is:

$$root(i) = 2 \times i - ones(i) \tag{3}$$

All the code numbers that the leaf node $LF_{cn(i)}$ brings are:

$$CNs(i) = \{root(i) - k | k = 0, 1, ..., root(i) - cn(i)\} \tag{4}$$

For example, when adding the 12th leaf node, we can get: $cn(12) = 20$, $root(12) = 22$ and $CNs(12) = \{22, 21, 20\}$.

Insertion Algorithm. The insertion algorithm does not need to read any data from disk and only root nodes of PISA are in memory. We use a stack to maintain all the root nodes. When the top two root nodes in the stack meet the combining conditions, we pop them from the stack, generate a new node by calculating a new aggregation value, and then push the new node into the stack. Steps (I)∼(II) in Figure 4 show the changes of the stack. In Figure 4, when adding LF_{20}, we put LF_{20} into the stack. Firstly, LF_{20} and LF_{19} meet the combining conditions, so IN_{21} is generated. Then IN_{21} and IN_{18} are merged and IN_{22} is generated. The algorithm of combining trees is described in Algorithm 1.

Algorithm 2 shows the method of adding a new window summary into PISA. When we initialize the index, we set $max\_serial = 1$. Then max_serial will be updated by calling Algorithm 2. Line 1 uses Equation 2 to calculate the code number according to max_serial. In this process, the algorithm will call hamming weight function $ones(i)$. When PISA meets combining conditions, line 6 will call $combine()$ to combine the trees in PISA.

Algorithm 1 combine();
combine trees that meet the combining conditions.

Input:
1. sn: the serial number;
2. $root\_stack$: all the root nodes.

Output:
successful or not;

1: calculate the code number of the largest root node according to Equation 3: $cn = 2 \times sn - ones(sn)$;
2: **while** $root\_stack.top.cn \neq cn$ **do**
3: pop up two root nodes, LF_i (or IN_i) and LF_j (or IN_j), from the $root\_stack$, mark the window summaries in them are $S_{2^n k(t_i)}$ and $S_{2^n k(t_j)}$;
4: generate new internal node IN_{j+1}, which $IN_{j+1}.cn = j+1$, and $IN_{j+1}.S = \mathbb{A}(S_{2^n k(t_i)}, S_{2^n k(t_j)})$;
5: $IO.write(IN_{j+1})$;
6: push IN_{j+1} into $root\_stack$;
7: **end while**
8: **return** successful.

5. QUERYING USING PISA

This section presents how to query the aggregation result based on PISA. Because PISA is persistent on disk, it is expensive to traverse the trees by reading all internal nodes. Therefore, we need to find a way to access those nodes which are part of the result without accessing other nodes.

5.1 Splitting a query range

A time series $D_{(qs,qe)}$ can be split into 3 parts: $D_{(s,e)}$ - its aggregation result can be read directly from the index; $\{p|p \in D_{(qs,s-1)}\}$ and $\{p|p \in D_{(e+1,qe)}\}$, which are incomplete windows, and their aggregation values need to be calculated on site.

Firstly, we use $start(W)$ and $end(W)$ to represent the start and end time of window W. Formally,

$$start(W) = t, \quad s.t. \quad p_t \in W \wedge (\nexists s < t \wedge p_s \in W)$$
$$end(W) = t, \quad s.t. \quad p_t \in W \wedge (\nexists e > t \wedge p_e \in W)$$

Secondly, we define the predecessor and successor of window $W_{k(s)}$ and data point p_t as follows. The predecessor of window $W_{k(s)}$ is the window that is adjacent to $W_{k(s)}$ but its start time is less than $W_{k(s)}$. Formally, the predecessor of window $W_{k(s)}$ is:

$\lhd(W_{k(s)}) = W_{k(s')}, s.t. \exists s', e' \in T \wedge start(W_{k(s')}) = s' \wedge |D_{(s',e')}| = k \wedge D_{(s',s)} - D_{(s',e')} - \{p_s\} = \emptyset$.

Algorithm 2 insert();
insert a new window summary (leaf node) into PISA.

Input:
1. max_serial: the maximal serial number in the index;
2. $S_{k(t_i)}$: the new window summary;
3. $root\_stack$: all the root nodes.

Output:
updated max_serial;

```
calculate the code number cn of the new leaf node according to Equation 2;
generate new leaf node LF_cn, LF_cn.SN = max_serial, LF_cn.S = S_{k(t_i)}, LF_cn.CN = cn;
IO.write(LF_cn);
push LF_cn into root_stack;
if max_serial is even then
    combine(max_serial,root_stack);
end if
max_serial = max_serial + 1;
return max_serial.
```

Similarly, the successor of window $W_{k(s)}$ is:
$\triangleright(W_{k(s)}) = W_{k(s')}, s.t. \exists s', e' \in T \wedge start(W_{k(s')}) = s' \wedge |D_{(s',e')}| = k \wedge D_{(s,s')} - D_{(s,e)} - \{p'_s\} = \emptyset$.

Thirdly, we define the predecessor and successor of a data point. If p_t is the last point in a window W, we define the W as the predecessor of point p_t, otherwise the predecessor of p_t is equal to the predecessor of W. Formally, the predecessor window of the data point p_t is:

$$\triangleleft(p_t) = \begin{cases} W_{k(s)}, & if \quad p_t \in W_{k(s)} \wedge t = end(W_{k(s)}) \\ \triangleleft(W_{k(s)}), & elseif \quad t \in [start(W_{k(s)}), end(W_{k(s)})), \end{cases}$$

Similarly, the successor of the data point p_t is:

$$\triangleright(p_t) = \begin{cases} W_{k(t)}, & if \quad p_t \in W_{k(t)} \wedge t = start(W_{k(s)}) \\ \triangleright(W_{k(s)}), & else \quad s.t. \quad t \in (W_{k(s)}, W_{k(e)}]. \end{cases}$$

Using the above definitions, we describe the splitting method as following:

Splitting Method. Given a query $Q_{\mathbb{A}}(qs, qe)$, we can split the query range into 3 parts:
(1) $Q_{\mathbb{A}left} = D_{(qs,start(\triangleright(p_{qs})))}$-$\{p_{start(\triangleright(p_{qs}))}\}$,
(2) $Q_{\mathbb{A}index} = D_{(start(\triangleright(p_{qs})),end(\triangleleft(p_{qe})))}$, and
(3) $Q_{\mathbb{A}right} = D_{(end(\triangleleft(p_{qe})),qe)}$-$\{p_{end(\triangleleft(p_{qe}))}\}$.

For example, in Figure 5, given a query $Q_{\mathbb{A}}(qs, qe)$, in which $t_{1e} < t_{qs} < t_{2s}$ and $t_{11e} < t_{qe} < t_{12s}$, there are $start(W_{t_2}) = t_{2s}$, $end(W_{t_{11}}) = t_{11e}$, $\triangleleft(p_{qe}) = W_{t_{11}}$ and $\triangleright(p_{qs}) = W_{t_2}$. We can split the time series $D_{(t_{qe},t_{qs})}$ into 3 parts: $D_{(t_{qs},t_{2s-1})}$, $D_{(t_{2s},t_{11e})}$ and $D_{(t_{11e+1},t_{qe})}$. We can get the aggregation value of $D_{(t_{2s},t_{11e})}$ by PISA while the other 2 parts have to be calculated on site.

5.2 Leftmost leaf node

After splitting a query range into 3 parts, we need to find the nodes which contain the aggregation values of $Q_{\mathbb{A}index}$ from the index.

We define the leaf nodes set that an internal node IN can **cover** as: when Considering IN as a root node of a tree, all the leaf nodes in the tree. The **leftmost leaf node** in the tree is the leftmost leaf node that IN can cover. To describe how to get the aggregation value by the index, we need two formulas: (1)given a leaf node whose serial number is even, getting all the code numbers of internal nodes which the leaf node brings and the depths of these nodes, (2) given an internal node, getting the leftmost leaf node which the internal node can cover.

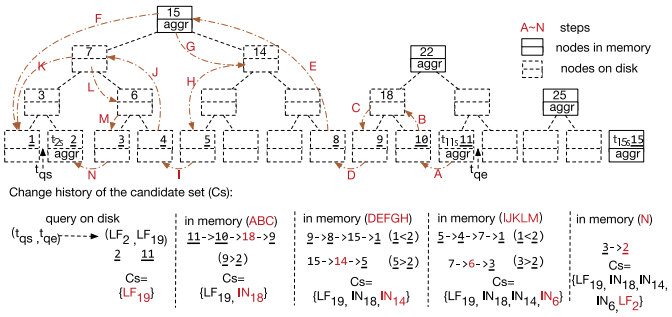

Figure 5: Example of query process by the index

Suppose i is a serial number, the corresponding leaf node is $LF_{cn(i)}$ and $cn(i)$ is the code number, which can be calculated by Equation 2. The code number set $CNs(i)$ of internal nodes which $LF_{cn(i)}$ brings can be calculated by Equation 4. If i is even, the depth of internal node IN_j ($j \in CNs(i)$) is:

$$depth(j) = j - cn(i) \tag{5}$$

The leftmost leaf node that IN_j can cover is:

$$\dashv(IN_j) = LF_{j+2-2^{(depth(j)+1)}} \tag{6}$$

For example, in Figure 5, all internal nodes which can cover the leaf node LF_{12} ($i = 8$) are IN_{15}, IN_{14}, IN_{13} and IN_{12} (i.e., LF_{12}), so $CNs(8) = \{15, 14, 13, 12\}$. Then $depth(15) = 3$ and $depth(14) = 2$. $\dashv(IN_{15}) = 1$, $\dashv(IN_{14}) = 8$ and $\dashv(IN_{13}) = 11$.

Finally, we define the predecessor and successor of a leaf node. Suppose leaf node LF stores $S_{k(s)}$, then:

$$\begin{aligned} \triangleleft(LF) &= LF', \quad s.t. LF'.SN = LF.SN - 1 \\ \triangleright(LF) &= LF', \quad s.t. LF'.SN = LF.SN + 1 \end{aligned} \tag{7}$$

5.3 The Query Algorithm

We first show a query example for illustration and then formulate the algorithm. When we query the maximal value of $D_{(t_{qs},t_{qe})}$ in Figure 5, we need to read data from disk to find LF_2 and LF_{19} (SN=2, 11), which are $\triangleright(p_{t_{qs}})$ and $\triangleleft(p_{qe})$. Then we can get the serial number and code number of the two leaf nodes. We do not need to read from disk anymore until we get the candidate set.

The text $A \sim N$ represent the query process. Firstly, we add LF_{19} into the candidate set Cs. $\triangleleft(LF_{19}) = LF_{17}$ according to Equation 7. Then we use LF_{17} (SN=8) to find out IN_{18}. IN_{18} can cover LF_{16} (SN=9>2). We add IN_{18} into Cs (steps ABC). $\triangleleft(LF_{16}) = LF_{12}$ (SN=8). We try IN_{15} to IN_{13} to check which is the first node that can not cover LF_2. Then we add IN_{14} into Cs (steps D∼H). Similar, we add IN_6 and LF_2 into Cs (steps I∼N). Finally, we get $C_s = \{LF_{19}, IN_{18}, IN_{14}, IN_6, LF_2\}$.

Notice that in all the steps above, A∼N, do not require reading data from disk, so it is highly efficient. If the index is a traditional segment tree, internal nodes such as IN_{15}, IN_7 and IN_3 have to be read from disk.

To get the final aggregation value of $Q_{A(t_{qs},t_{qe})}$, we need 3 steps: (1) read the aggregation values of C_s from the index; (2) scan the data points in $D_{(t_{qs},t_{2s-1})}$ and $D_{(t_{11e+1},t_{qe})}$ and

Algorithm 3 Query algorithm

Input:
$Q_{\mathbb{A}(qs,qe)}$: the query range $D_{(qs,qe)}$ and the aggregation operation $\mathbb{A}$.
Output:
aggregation value;
1: use **splitting method** to split the query range int 3 parts: $Q_{\mathbb{A}left}$, $Q_{\mathbb{A}index}$ and $Q_{\mathbb{A}right}$. And mark $Q_{\mathbb{A}index}$ as $Q_{\mathbb{A}(s,e)}$;
2: use $IO.batch\_get(start(\triangleright(p_{qs})), start(\triangleleft(p_{qe})))$ to get $LF(s)$ and $LF(e)$;
3: set $Node_{candidate} = \emptyset$;
4: **if** $Q_{\mathbb{A}(s,e)} \neq \emptyset$ **then**
5: set $LF = LF(e)$; // begin with the last leaf
6: **if** $LF.SN$ is even **then**
7: add $LF.name$ into $Node_{candidate}$;
8: set $LF = \triangleleft(LF)$ and goto line 4;
9: **else**
10: get all the code numbers CNs of the internal nodes which LF brings by Equation 4;
11: **for** each cn in CNs **do**
12: **if** IN_{cn} can not cover $LF(s)$ **then**
13: add cn to $Node_{candidate}$;
14: set $LF = \triangleleft(\dashv(IN_{cn}))$;
15: skip the loop and goto line 4;
16: **end if**
17: **end for**
18: **end if**
19: **end if**
20: use $IO.batch\_get(Node_{candidate})$ to get nodes from the store engine in batch;
21: calculate the aggregation of $Q_{\mathbb{A}left}$, $Q_{\mathbb{A}right}$ by reading all the original data points;
22: calculate the final aggregation;
23: **return** the final aggregation.

then calculate the aggregation values; and (3) get the final result by merging the aggregation values in (1) and (2).

Algorithm 3 describes how to query on the index in general. Data is read from disk in lines 2, 20 and 21. Line 2 needs to read only two leaf nodes. The number of data points that line 21 reads depends on the window size. The maximum is less than 2 times the window size. Line 20 needs to read at most $log_2(2n/k)$ nodes, in which n is the number of the data points and k is the window size.

The complexity of the query process is $O(log_2(n/k) + k)$. Because the window size is k, there are n/k leaf nodes in total. By adding at most n/k virtual leaf or internal nodes, we can convert the forest to a complete binary tree (like Figure 6). The virtual tree has at most $2n/k$ leaves. In the virtual segment tree, the maximal node which a query needs is $log_2(2n/k)$. In Algorithm 3, for each loop, we can get one candidate node. Therefore, the complexity of the algorithm is $O(log_2(n/k))$. Because we need to read at most $2k$ data points additionally, the complexity is $O(log_2(n/k)) + k$.

5.4 Cassandra and MySQL Implementations

In this subsection, we introduce how to use Cassandra and MySQL to support write operation $IO.write()$, and read operations $IO.batch\_get()$, $IO.get()$, $IO.gt()$ and $IO.ls()$. Storing PISA by Cassandra's standard interfaces rather than a local file has the following advantages: Cassandra gives the index the abilities of fault tolerance and distribution (though we do not discuss how to build a distribution version of PISA in this paper).

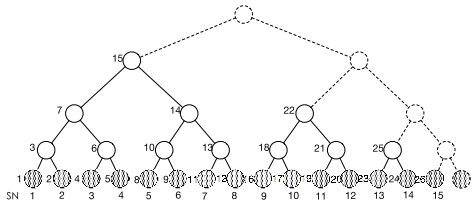

Figure 6: Convert a forest to a complete binary tree

We create a table to store the data of PISA by using "*create table PISA (ts_name string, start_time long, value bytes, primary key (ts_name, start_time))*". In the table, *ts_name* represents a time series. *start_time* represents the start time of a window (leaf node) or the code number of an internal node. *value* stores the aggregation value of the window. To access nodes quickly, we create index on *ts_name* and *start_time* on MySQL. Cassandra does not need the declaration because for each *ts_name*, the data in the table is ordered by the *start_time* by default.

The $IO.write()$ operation can be implemented by a simple "*insert ... into..*" statement with SQL or CQL3. The $IO.get()$ operation can be achieved by a"*select*" statement with a "*where start_time=...*" clause while $IO.batch\_get()$ operation can be finished by "*where start_time in (...)*" clause. $IO.gt()$ and $IO.ls()$ can be implemented by "*where start_time >= (<=) ... (order by start_time desc) limit 1*".

As a whole, PISA can be stored by some basic SQL (CQL3 in Cassandra) statements. That is to say, PISA is highly adaptable and we can implement PISA easily in most database systems.

6. EXPERIMENTS

6.1 Experiment settings

We have implemented PISA both on a NoSQL database (Cassandra) and a traditional relational database (MySQL).

Cassandra version 2.11, the latest stable version for now, is used in our experiments. It runs on a commodity PC that has 8 cores of Inter i7 with 3.4GHz, 16GB memory and 7200 rpm HDD. We use the Datastax Java driver and CQL3 to read and write data from/to Cassandra. For MySQL, version 5.7 is used with MySQL JDBC driver to read and write data. We have generated 2^{34} (16 billions) data points for our experiments. In Cassandra, 2^{34} points has taken approximately 80GB of disk space [1]. The experiments include:

(1) Evaluate the memory cost;
(2) Evaluate the insertion speed of PISA;
(3) Evaluate the impact of window size;
(4) Evaluate the query speed of PISA.

In the second (insertion speed) and fourth (query speed) experiments, we have implemented 6 approaches in Cassandra and use the first 5 approaches as baselines. These approaches include:

[1]Cassandra uses LZ4 to compress data by default. The uncompressed data size is 16G * 8 Byte (long or double) *3 (time, value, version) = 384GB theoretically.

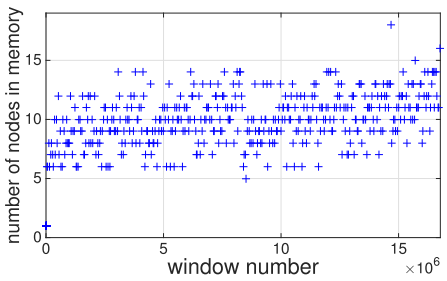

Figure 7: Memory cost with the increment of window number

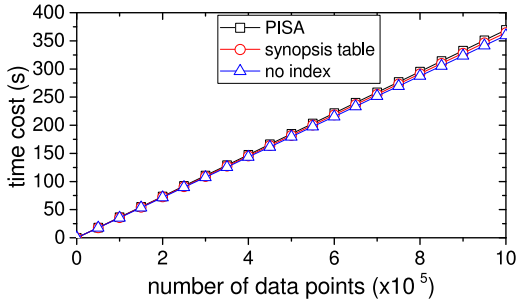

Figure 8: Insert speed for different indexes

(1) Read the original data points directly from disk and then calculate the aggregation value in memory;
(2) Use synopsis table;
(3) Use Parsimonious Temporal Aggregation (PTA) [6];
(4) Use SB-tree [23];
(5) Use Balanced-tree [13];
(6) Use PISA.

Note that PTA does not generate an exact aggregation result. To accelerate the query, it allows ϵ-error for the result. When $\epsilon = 0$, the approach is degenerated to the synopsis table approach. When $\epsilon = 1\%$, the approach only needs to scan about 20% of the synopsis table.

SB-tree combines segment tree and B-tree together. In our experiment, we define that each B-tree node has 10 children.

Balanced-tree uses a self-balancing tree to implement the segment tree structure, so the segment tree can be constructed online. Its query performance is similar to the traditional segment tree.

For fairness, data are stored in Cassandra rather than in memory throughout all experiments (i.e., for all the approaches above). Besides, after inserting all the data points, we restart the operation system to make sure PISA is not cached in memory by the operation system or databases.

6.2 Memory cost

One of the key features of PISA is the dependence-free memory. Figure 7 shows the number of root nodes in the memory when we build the index. The result is consistent with Equation 1. We can find that the root number changes in a sawtooth shape and increases slowly. For 1.6×10^7 windows, there are at most 17 root nodes in the memory. For each root node, the memory size is related to the aggregation information in the node. If a PISA only accounts for the maximal value, there is only one float value in the node. If the index includes the maximum value, minimum value, average value, we need to store 3 floats in each node. If the index wants to know the variance more, we have to store the quadratic sum value in the root node additionally. Therefore, the actual memory cost depends on what aggregation query we want to support.

6.3 Insertion speed

We set the experiments on Cassandra to evaluate the insertion speed of PISA and to compare with other approaches. In Figures 8, we inserted 1 million data points. The index generated one leaf node and several internal nodes per 100 data points (i.e., 100 points per window).

In Cassandra, the data load program without index takes 3.6×10^5 milliseconds while it takes 3.7×10^5 milliseconds when applying the PISA. That is, the PISA only occupies 2.7% time cost in the data load program. In summary, the index has little effect on the write speed. MySQL has a similar result.

The insertion speed of the 2nd approach, synopsis table, is slightly faster than that of PISA. The 3rd approach is an offline building method, so we do not consider it here.

The insertion speeds of the 4th and 5th approaches are not drawn in Figure 8. This is because they are too slow when we build them on disk rather than in memory. For example, when we use an AVL tree to implement the 5th approach, the insertion speed is about 1/400 of the speed without any index. It is reasonable because for each point, the approach generates a new leaf node. When a new leaf node is generated, its parent (and grandparent and so on) needs to be updated because the aggregation value changes. This operation costs much less when the tree is in memory. However, if the tree is on disk, the update operations create a big time cost. PISA uses a form of forest and a stack to solve the issue, so it never needs to update data on disk. As a result, it is far faster than the 4th and 5th approaches.

6.4 Impact of window size

Insertion. Window size impacts the insertion speed. This is because the number of times for calling $IO.write()$ is between$[n, 2n]$, in which n is the number of windows. Because n is equal to $\frac{\text{the number of total points}}{\text{window size}}$, the larger that window size is, the faster the insertion speed is.

If the window size is small (e.g., size is 1), the insertion speed will decline seriously. A solution to speed up the insertion is by using $IO.batch\_write()$ rather than $IO.write()$. The former can write several nodes on disk in batch. In practice, it is faster than the latter and many NoSQL databases support the operation. A side effect is that we have to save the batched nodes in memory temporarily.

Query. Window size also impacts the query speed, because the query complexity is $O(n/k + k)$, in which k is the window size. In this experiment, we change the window size from 10 to 1000. Then we generate query ranges which contain about 10 million data points.

Figure 9 shows that a larger window size does not certainly save the time cost for query. For example, PISA is the fastest when $k = 100$ in our Cassandra experiment (200 or 400 in MySQL). We conjecture that the reason is if the window size is large, PISA needs to read too many data points in $Q_{\mathbb{A}left}$ and $Q_{\mathbb{A}right}$. For example, when $k = 100$, PISA needs to read at most 198 data points while PISA has to read at most 19998 data points if $k = 10000$.

In the following experiments, we set window size $k = 100$.

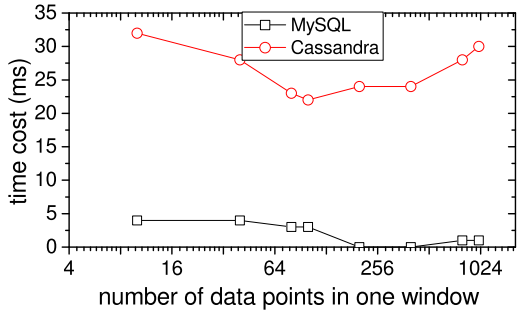

Figure 9: Time costs of PISA for different window sizes

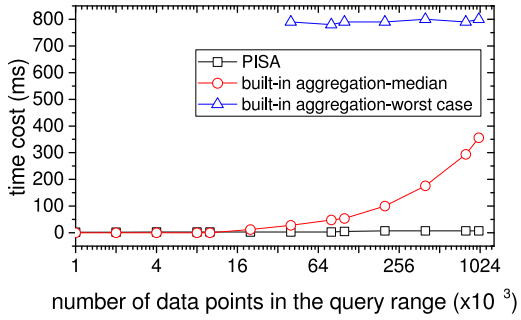

Figure 10: Time costs of aggregation indexes in MySQL

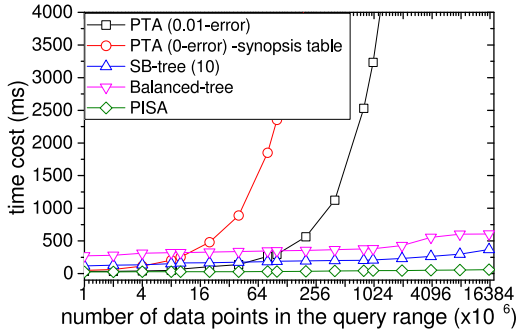

Figure 11: Time costs of aggregation indexes in Cassandra

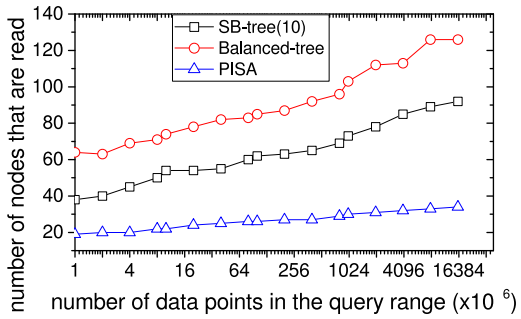

Figure 12: number of nodes needed to read from disk by different aggregation indexes in Cassandra

6.5 Query performance on MySQL

MySQL supports aggregation functions (e.g., max(), min(), avg()). In this experiment, we compare the time costs of the following two approaches:

(1)The built-in aggregation function in MySQL;

(2) PISA index.

The configurations of MySQL are as follows. To get the fastest response time, we declare the MySQL build-in index on the data points by using "create index on points(value)" SQL statement. Besides, when repeating the experiment on MySQL, we find that the time cost is always 0 when we repeat a query (with the same condition) for the second time. The reason we conjecture is that MySQL caches recent queries. Therefore, we close the cache by setting "SET SESSION query_cache_type = OFF". Then we can repetitively run queries to calculate the average time cost.

Figure 10 shows the result of querying the aggregation by MySQL built-in aggregation function directly and PISA. In this experiment, we only test about 1 million data points. Firstly, the time cost of the built-in index increases quickly when the query range increases. Secondly, we notice that when querying ranges sometime, the time costs are high and these cases can be reproducible. 'built-in aggregation-worst case' shows the time costs. Thirdly, PISA shows a faster speed than the built-in index, e.g., 7ms:300ms for querying 1 million points.

6.6 Querying performance on Cassandra

Cassandra does not support any aggregation operation except for *count*() function which returns the number of data points when given a time range. Besides, the *count*() function will be timeout when there are more than 10 million points (timeout threshold is 50s).

In this experiment, we use the 6 approaches in Section 6.1 to query and compare their time costs. For each approach, we generate a set of time range sizes: from the range which has about 1 million points to the range which has about 16 billion points. The start time and end time of a range are generated randomly.

The 1st approach is far slower than other approaches. In our experiment, it takes about 7 seconds to query 1 million data points, and takes about 16 seconds to query 2.5 million data points. The time cost is in proportion to the number of data points. According to the results, we consider this approach to be unsuitable for ad-hoc query requirements where the number of data points is larger than 1 million. By the way, the time cost of this approach is similar to the *count*() function in Cassandra.

Figure 11 shows the comparison between the 2nd∼6th approaches. In our experiment, all the indexes are stored in Cassandra rather than in memory. Firstly, because PTA and synopsis table do not use tree based structure, their performances decline quickly. Secondly, the performances of SB-tree, Balanced-tree and PISA are better. Thirdly, PISA has the best performance in these approaches.

The good performances of SB-tree and Balanced-tree rely on the low latency of read operation in Cassandra. In fact, SB-tree and Balanced-tree need more read operations on disk than PISA.

To illustrate the inherent difference among SB-tree, Balanced-tree and PISA, we count the number of nodes that these three approaches need to read from disk. Figure 12 shows the results. Firstly, Balanced-tree needs to read the most nodes. Many of these nodes are just for deciding which children need to be accessed, while these nodes themselves are not parts of the final aggregation result. Moving on,

SB-tree is better than Balanced-tree but worse than PISA. SB-tree has the same problem with Balanced-tree. However, because the depth of SB-tree is less than that of Balanced-tree, the number of unnecessary nodes is less than the latter as well. PISA needs to read the least nodes. All the nodes that PISA reads are parts of the final aggregation value.

The number of nodes that PISA needs to read is about 1/2∼1/6 of SB-tree or Balanced-tree, and the time cost takes 1/10 or less (e.g., PISA needs 55ms while Balanced-tree needs 605ms when we query 8 billion points). This is due to the fact SB-tree and Balanced-tree traverse data level by level. For each read operation (i.e., $IO.get()$), we can only get one internal or leaf node. As a result, we need too many read operations. On the other hand, as mentioned in Section 5.3, PISA can get the candidate set in memory, and then read nodes in batch (via the two $IO.batch\_get()$ operations in Algorithm 3).

7. CONCLUSION

Aggregation operations are important in time series data management. In this paper, we have proposed PISA, an index for aggregation query on billions of data points in a time series. To implement the PISA, we split the time series into (fixed size) windows and use extended segment trees to organize them. Different to traditional segment trees, PISA can be built online and its nodes store different information. Then we proposed a new algorithm to query data from the trees which avoids reading unnecessary nodes from disk. Besides, for time series with billions of data points, PISA only consumes several hundred bytes of memory.

We have implemented PISA on both Cassandra and MySQL. Given an arbitrary time range, PISA returns the aggregation result in milliseconds even though the time range has billions of data points. Experiments have shown that (1) PISA has better performance for the insertion operation than current segment tree based indexes, and (2) PISA provides lower latency for the aggregation query than many existing approaches.

Currently, PISA supports aggregations such as $count()$, $max()$, $average()$, $var()$. PISA can also approximate the aggregation values that require sorting all the data points, e.g., $mean()$. Our ongoing work is to further optimize this kind of aggregations.

8. ACKNOWLEDGMENTS

This work was supported by Project 61325008 (NSFC), and the National Key Technology Support Program (No. 2015BAH14F02).

9. REFERENCES

[1] V. Abramova and J. Bernardino. NoSQL databases: MongoDB vs Cassandra. In *C3S2E*, pages 14–22. ACM, 2013.

[2] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental sliding-window computations for large-scale data analysis. *CITI, Universidade Nova de Lisboa, Lisbon*, 2012.

[3] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. More geometric data structures. *Computational Geometry: Algorithms and Applications*, pages 219–241, 2008.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[5] J. Gamper, M. Böhlen, and C. S. Jensen. Temporal aggregation. In *Encyclopedia of database systems*, pages 2924–2929. Springer, 2009.

[6] J. Gordevičius, J. Gamper, and M. Böhlen. Parsimonious temporal aggregation. *VLDBJ*, 21(3):309–332, 2012.

[7] M. Kornacker and Behm. Impala: A modern, open-source SQL engine for Hadoop. CIDR, 2015.

[8] W. Lehner, B. Cochrane, H. Pirahesh, and M. Zaharioudakis. Applying mass query optimization to speed up automatic summary table refresh. In *ICDE*, 2001.

[9] C. Li, J. Chen, C. Jin, R. Zhang, and A. Zhou. MR-tree: an efficient index for MapReduce. *IJCS*, 27(6):828–838, 2014.

[10] Y. Li, G. Kim, L. Wen, and H. Bae. Mhb-tree: A distributed spatial index method for document based nosql database system. In *UITA*, pages 489–497. Springer, 2013.

[11] G. Mahlknecht, A. Dignös, and J. Gamper. Efficient computation of parsimonious temporal aggregation. In *ADIS*, pages 320–333. Springer, 2015.

[12] D. P. Mehta and S. Sahni. *Handbook of data structures and applications*. CRC Press, 2004.

[13] B. Moon, I. F. V. Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *TKDE*, 15(3):744–759, 2003.

[14] S. B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1):147–175, 1989.

[15] P. Nikolov and G. Pierre. Aggregate queries in NoSQL cloud data stores. *University Amsterdam*, 2011.

[16] D. Pallett. Improving query performance of holistic aggregate queries for real-time data exploration. 2014.

[17] Z. Parker, S. Poe, and S. V. Vrbsky. Comparing NoSQL MongoDB to an SQL DB. In *ACMSE*, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[18] D. Peng, K. Duan, and L. Xie. Improving the performance of aggregate queries with cached tuples in MapReduce. *IJDTA*, 6(1):13–24, 2013.

[19] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. DB2 with BLU acceleration: So much more than just a column store. *VLDB*, 6(11):1080–1091, 2013.

[20] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. DB2 with BLU acceleration: So much more than just a column store. *VLDB*, 6(11):1080–1091, 2013.

[21] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *VLDB*, 8(7):702–713, 2015.

[22] V. K. Wei. Generalized hamming weights for linear codes. *IEEE TIT*, 37(5):1412–1418, 1991.

[23] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDBJ*, 12(3):262–283, 2003.