# Distributed In-memory Analytics for Big Temporal Data

Bin Yao[1,2], Wei Zhang[1,3], Zhi-Jie Wang[4], Zhongpu Chen[1], Shuo Shang[5], Kai Zheng[6], and Minyi Guo[1]

[1] Shanghai Jiao Tong University, Shanghai, China.
[2] Guangdong Key Laboratory of Big Data Analysis and Processing
[3] Guangdong Province Key Laboratory of Popular High Performance Computers
[4] School of Data and Computer Science, Sun Yat-Sen Univeristy, Guangzhou, China
[5] Extreme Computing Research Center, KAUST, Mecca, Saudi Ara
[6] School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China
yaobin@cs.sjtu.edu.cn, zhangweilst@sjtu.edu.cn,
wangzhij5@mail.sysu.edu.cn, shuo.shang@kaust.edu.sa,
zhengkai@uestc.edu.cn, guo-my@cs.sjtu.edu.cn

**Abstract.** The temporal data is ubiquitous, and massive amount of temporal data is generated nowadays. Management of big temporal data is important yet challenging. Processing big temporal data using a distributed system is a desired choice. However, existing distributed systems/methods either cannot support native queries, or are disk-based solutions, which could not well satisfy the requirements of high throughput and low latency. To alleviate this issue, this paper proposes an Inmemory based Two-level Index Solution in Spark (ITISS) for processing big temporal data. The framework of our system is easy to understand and implement, but without loss of efficiency. We conduct extensive experiments to verify the performance of our solution. Experimental results based on both real and synthetic dataset consistently demonstrate that our solution is efficient and competitive.

**Keywords:** Big temporal data, distributed in-memory analytics, Apache Spark, temporal queries

## 1 Introduction

Temporal data management has been studied tens of years and has gained increasingly interest in recent years [17, 26], due to its widely applications. For example, users may wish to investigate the demographic information of an administrative region (e.g., California) at a specific time (e.g, five years ago). Querying a historical version of the database (like above) is usually referred to as time travel [11, 5, 28]. As another example, in the quality assurance department users may wish to analyze how many orders are delayed as a function of time, thereby querying all historical versions of the database over a certain time period. Queries like mentioned above is usually called temporal aggregation [10, 20, 19].

In the existing literature, there are already a large bulk of papers addressing the problems of time travel and temporal aggregate queries (see e.g., [11, 5, 28, 20, 11, 21, 25]). Yet, most of prior works focused on developing single-machine-based solutions, and few attention has been made on developing distributed solutions for handling big temporal data. Nowadays, various *apps*, e.g., web apps and also Internet of things (IOT) apps, generate massive amount of temporal data. It is urgent need to efficiently process big temporal data. In particular, it is challenging to handle such a large volume of temporal data in traditional database systems. Clearly, processing such a large volume of temporal data using a distributed system should be a good choice. Recently, distributed temporal analytics for big data have been also investigated (e.g., [39, 9]). These works share at least two common features: (i) they are distributed *disk-based* temporal analytics; and (ii) time travel and temporal aggregate queries are not covered in their papers. With the surging data size, these solutions could not well meet the demand of high throughput and low latency.

Spark SQL [37] is such an engine, which extends Spark (a fast distributed *in-memory* computing engine) to enable us to query the data inside Spark programs. To support distributed in-memory analytics for big temporal data with high throughput and low latency, this paper proposes an In-memory based Two-level Index Solution in Spark (ITISS). To the best of our knowledge, none of existing big data systems (e.g., Apache Hadoop, Apache Spark) provides native support for temporal data queries, and none of prior works develops distributed in-memory based solution for processing time travel and temporal aggregation over big temporal data. To summarize, the main contributions of our work are as follows:

- We propose a distributed in-memory analytics framework for big temporal data. Our framework is easy to understand and implement, but without loss of efficiency.
- We present targeted algorithms for answering time travel and temporal aggregation queries, by fully utilizing the proposed framework that adopts a two-level index structure.
- We implement our framework in Apache Spark, and extend the Apache Spark SQL to support declarative SQL query interface that enables users to perform the complex tasks with a few lines of SQL statements.
- We conduct a comprehensive experimental evaluation for our proposed solution, using both real and synthetic temporal data. The experimental results consistently demonstrate the efficiency and competitiveness of our proposal.

The rest of this paper is organized as the following. Section 2 formulates our problem. Section 3 presents the framework for big temporal data, including a distributed indexing structure, the query procedures, and the implementation details based on Apache Spark. We present the experimental evaluation in Section 4. Section 5 reviews prior works most related to ours, and Section 6 concludes this paper.

**Table 1.** Frequently Used Notations

| Notation | Description |
|---|---|
| $D$ | a temporal dataset |
| $t_i$ | the $i$-th temporal record of $D$ |
| $I_p$ | a partition interval |
| $v$ | snapshot version number of temporal database |
| $Q_e$ | time travel exact match query |
| $Q_r$ | time travel range match query |
| $Q_a$ | temporal aggregation operation |
| $g$ | a temporal aggregation function, e.g. *SUM*, *MAX* etc. |

## 2 Problem Definition

Specifically, this paper attempts to achieve two representative operations (i.e., *time travel* and *temporal aggregation*) over *temporal data* in distributed environments. Nevertheless, our framework and algorithms described later can be easily extended to support other operations (e.g., *temporal join*) and other data (e.g., *bitemporal data* [7]). In what follows, we formally define our problems. (For ease of reference, Table 1 lists the frequently used notations.)

Given a temporal dataset $D$ containing $|D|$ temporal records $\{t_1, t_2, ..., t_{|D|}\}$. Each record $t_i$ ($i \in [1, |D|]$) is a quadruple in the form of $(key, value, start, end)$, where *key* corresponds to the id of the record, *start* and *end* are the starting and ending timestamps of a time interval in which the record is alive. Further, given a version (or timestamp) $v$ and a record $t_i$, we say that record $t_i$ exists in version $v$ (i.e., record $t_i$ is alive in version $v$), if and only if $v \in [t_i.start, t_i.end)$.

Time travel establishes a consistent view for the history of a database, and it is one of the most significant temporal operations in temporal databases. Here we address two widely used time travel operations, i.e., *time travel exact match* and *time travel range match*. Both of operations can support querying the past version of a database. Their major difference is that the input of *exact-match query* uses a specific value, while the input of *range query* uses a given range [5]. Specifically, their formal definitions are formulated below.

**Definition 1 (Time travel exact-match query).** *Given a time travel exact query $Q_e = \{key, v\}$, we are asked to retrieve the record (denoted as $\theta$) from $D$ such that,*

$$\theta = \{t_i \in D \mid t_i.key = key \wedge t_i.start \leq v \wedge v < t_i.end\}.$$

As an example, consider a simple temporal database with 7 temporal records as shown in Fig. 1. When $Q_e = \{21, v_1\}$, the query return $t_3$; in contrast, when $Q_e = \{21, v_2\}$, the query returns $\emptyset$.

**Definition 2 (Time travel range query).** *Given a time travel range query $Q_r = \{start\_key, end\_key, v\}$, we are asked to retrieve a set $\theta$ of records from $D$ such that,*

$$\theta = \{t_i \in D \mid start\_key \leq t_i.key \wedge t_i.key \leq end\_key \wedge t_i.start \leq v \wedge v < t_i.end\}.$$

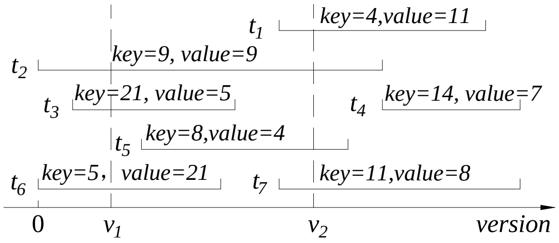


**Fig. 1.** Temporal Aggregation

As an example (see also Fig. 1), when $Q_r = \{7, 22, v_1\}$, the query returns $\{t_2, t_3\}$; in contrast, when $Q_r = \{7, 22, v_2\}$, the query returns $\{t_2, t_5, t_7\}$.

Temporal aggregation is a common operation in temporal database, and usually is challenging and expensive. Since temporal aggregation was introduced by [21], it has been heavily studied. In this paper we focus on aggregation (e.g., MAX, SUM) conducted at a specific timestamp. Formally, the temporal aggregation operation is defined as follows.

**Definition 3 (Temporal aggregation query).** *Given a temporal aggregation query $Q_a = \{g, v\}$ where $g$ is an aggregation function such as MAX, we are asked to return an aggregate value (denoted as $\theta$) based on $D$ such that,*

$$\theta = g\{t_i \in D \mid t_i.start \leq v \wedge v < t_i.end\}.$$

Consider also the example shown in Fig. 1. When $Q_a = \{MAX, v_1\}$, the query returns 21 (since $max\{9, 21, 5\} = 21$); in contrast, when $Q_a = \{SUM, v_1\}$, the query returns 32 (since 4+9+8+11=32).

Note that, compared with prior works, in this paper our focus is on big temporal data in distributed environments. As discussed in Section 1, a straightforward implementation based on existing distributed systems is inefficient and ineffective; in the next section we present our solution in detail.

## 3 Our Solution

In this section, we first describe the distributed processing framework. Then, we show how to achieve time travel and temporal aggregation queries based the proposed framework. Finally, we discuss the implementation details of deploying the framework onto the classic distributed computing engine — Apache Spark.

### 3.1 System Framework

At a high level, our framework consists of three parts: (i) Partition unit. It is responsible for partitioning all data into distributed (slave) nodes. Usually, we should guarantee each node having roughly same size of data, in order to

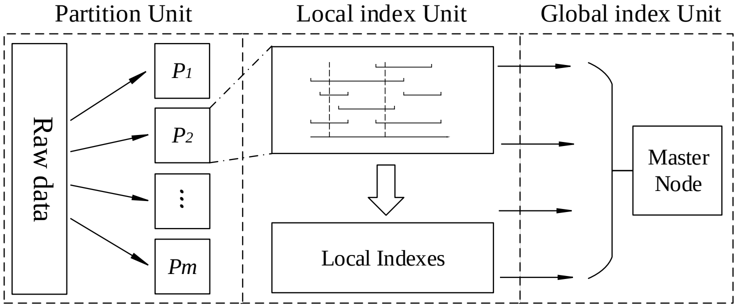
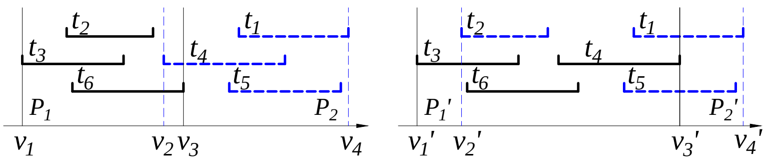


**Fig. 2.** The architecture of our system framework.

keep the *load balance.* (ii) Local index unit. Within each partition, the local indexes are maintained to avoid a "full" scanning, and so may help us boost the query efficiency. In addition, each partition also maintains the *partition intervals* (explained later) for the global index construction. And (iii) global index unit. In the master node a global index is designed to prune "unpromising" partitions in advance. This can avoid checking each (individual) partition, and thus may help us reduce the CPU cost and/or network transmission cost. In our design, the master node collects all partition intervals from each partition in slave nodes, and then builds the global index based on the collected partition intervals. The architecture of our framework is shown in Fig. 2. It is easy to understand that our framework adopts a two-level indexing structure, which can avoid visiting irrelevant candidates (e.g., partitions and local records) as much as possible. Although the rational behind the framework is simple, it is definitely efficiency as demonstrated later. In what follows, we discuss important issues in each unit.

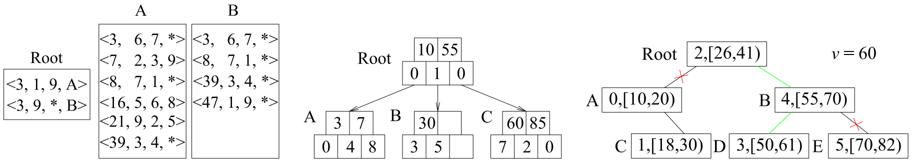
► *Partition method.* Typically, load balance is a desired goal when partitioning the general data. As to the temporal data, another desired goal is to minimum the overlap of partition intervals. To achieve these goals, in our design we partition the data by interval (known as *range partition*). As an example, assume one wants to partition six temporal records, shown in Fig. 3(a), into two partitions $P_1$ and $P_2$. He/she can first sort these temporal records by their intervals, obtaining a sorted records $(t_3, t_2, t_6, t_4, t_5, t_1)$. To balance the size of each partition, he/she



(a) Range partition method

(b) Hash partition method

**Fig. 3.** Different partition methods



(a) MVB-Tree (b) SB-Tree (c) Global Pruning

**Fig. 4.** Local index structure

can evenly split the sorted records into two. As a result, $P_1$ contains first three records $(t_3, t_2, t_6)$, and correspondingly $P_2$ contains $(t_4, t_5, t_1)$. This way, the partition interval of $P_1$ is $[v_1, v_3)$, and that of $P_2$ is $[v_2, v_4)$. In particular, the interval overlap of $P_1$ and $P_2$ is $v_3 - v_2$, which is the minimum overlap.

Note that, although using *hash* to partition the data is widely used for other data domain such as streaming data (since the data can be evenly allocated via this manner), it could be not appropriate for the context of our concern. The major reason is that partitioning in such a way could cause many overlaps (among partition intervals). For example, consider the temporal data shown in Fig. 3(b). After finishing hash partition, $P'_1$ contains $(t_3, t_4, t_6)$ and $P'_2$ consists of $(t_1, t_2, t_5)$. One can easily see that the interval overlap of $P'_1$ and $P'_2$ is $v'_3 - v'_2$, which is much larger than that of $P_1$ and $P_2$.

► *Local index method.* As mentioned earlier, the local index is used to manage the temporal data in each partition. In existing literature, there are already on-shelf index structures to support time travel queries such as *multiversion B-tree* [5] and *time-index* [11]. In our paper, we use multiversion B-tree (shorted as MVB-Tree) as a sample. For ease of understanding, Fig 4(a) shows an example of this index structure. The first entry of the root points to its leaf child $A$, which contains all the records that are alive from version 1 to 9 (excluded). In the leaf nodes, each entry represents a record, where $*$ means that this record is still alive now.

Also, there are already existing index structures (e.g., [35, 29]) to support temporal aggregation queries. Here we use the index (named SB-Tree) developed in [35] as a sample. The SB-Tree node is composed of two arrays, as illustrated in Fig 4(b). One of arrays stores the intervals, this array is used for pointing to the children nodes, and another stores the aggregate values. To calculate an aggregation using the SB-Tree, one can search the tree from the root to the leaf, and aggregate the values in its path.

Note that, in this paper, we mainly focus on how to leverage existing indexes to support distributed in-memory analytics (particularly, time travel and temporal aggregate queries) for big temporal data, and hence we care more about the design principle of the system, rather than addressing the limitations of existing index structures. In addition, although this paper adopts the MVB-tree and SB-tree, it is not compulsory to use these indexes. In other words, other on-shelf indexes, or more powerful indexes developed in the future can be also used in our framework.

```
Algorithm 1: ExactMatchQuery ($key, v$)
$R \leftarrow \varnothing$
$P \leftarrow$ GlobalPruning($v, r_g$)
foreach $p$ in $P$ do
    $root \leftarrow r_l$
    while $root$ is not leaf do
        $root \leftarrow$ child of $root$ whose route directs to $key$ and $v$
    end while
    if $key$ exists in $root$ then
        add $record$ containing $key$ to $R$
    end if
end foreach
return $R$
```

► *Global index method.* As discussed previously, the global index manages the partition intervals. Since each partition interval is a pair of version numbers, and is comparable by start value and length of the interval, naturally we can use the binary search tree to maintain partitions' interval information. Note that, for each partition in slave nodes, there are many time intervals. Nevertheless, we only use one *partition interval* for a partition. To understand, consider a simple example with three time intervals $\{[u_1, u_2], [u_3, u_4], [u_5, u_6]\}$ in a partition. Then, the partition interval is $[min\{u_1, u_3, u_5\}, max\{u_2, u_4, u_6\}]$. This way, each partition interval in the global index essentially corresponds to a partition in slave nodes. This implies that, in the query processing, if a partition interval can be pruned, then the corresponding partition can be pruned safely. Based on this intuition, in our design each node in the global tree maintains a key-value pair $< I_p, p_{id} >$, where $I_p$ and $p_{id}$ refer to the partition interval and its corresponding partition, respectively.

### 3.2 Query Processing

The query evaluation in our framework consists of two phases: (i) global pruning, and (ii) local look-up.

The first phase essentially is to fully utilize the global index and the version $v$ (in the query input) to prune "unrelated" partitions. To understand, consider an example shown in Fig. 4(c). Assume one wants to prune partitions that does not belong to version 60, he/she can traverse the global index to examine the partition interval. As a result, only two partitions (id=3 and id=4) can be regarded as the candidates. In contrast, the second phase mainly retrieves, in each candidate partition, the "qualified" records, based on the local indexes and part of query inputs. As an example, consider Fig. 4(a) and assume a time travel exact-match query $Q_e = \{key = 8, v = 6\}$; the local look-up first finds the entry that belongs to version 6 at the root node. Then, it checks the child $A$, in which we can find an entry with $key = 8$, and its valid time interval is $[1, *)$ containing

6. This completes the local look-up. In what follows, we cover detailed query algorithms for time travel and temporal aggregation queries.

▶ *Time travel queries.* We first discuss the time travel exact query, followed by the time travel range query. Algorithm 1 shows the pseudo-codes of the *time travel exact query*. Note that, Line 2 is used to perform global pruning, detailed in Algorithm 2. After finishing the global pruning at the master node, we obtain the `ids` of candidate partitions, which are stored in $P$. Then, the local look-up (Lines 2-15) retrieve the results in each partition; here local look-ups for all these candidate partitions are distributed to the cluster and executed in parallel. Note that, the algorithm for *time travel range query* is similar to Algorithm 1. The difference is that, we do not need to find the *entry* for the given key (Line 8). Instead, we maintain an array for *entries* that can direct to $[start\_key, end\_key]$, and then examine each block referenced by *entry* in *entries*. More details are shown in Algorithm 3.

▶ *Temporal aggregation queries.* When processing the temporal aggregation queries, the global pruning process is same to that for the time travel queries. Yet, the local look-up phase works in a different way. In brief, in each candidate partition, it first finds the *child* of the *root* so that the interval contains version $v$. If *child* is a leaf node, we just return aggregate value (denoted as $r$) in it. If not, we recursively find the aggregate value (denoted as $s$) of $v$ in *child*, and return aggregate value $r$ and $s$. The pseudo-codes are shown in Algorithm 4.

### 3.3 Implementation on Apache Spark

In Apache Spark the resilient distributed dataset (RDD) is fault-tolerant and can be stored in memory to support fast data reusing without accessing disk. In this section, we elaborate how to implement our framework in Apache Spark.

To support partition method suggested in Section 3.1, we extend Spark's **RangePartitioner**. Note that, Spark's **RangePartitioner** is developed for the general purpose data partition, it cannot effectively support partition via range. To achieve this function, we implement the comparision procedure for interval data format, and integrate it to Spark **RangePartitioner**.

**Algorithm 2:** GlobalPruning ($v$, *root*)

```
R ← ∅
if root ≠ null then
    if $v \in root.I_p$ then
        add root.id to R
    end if
    GlobalPruning(v, root.left)
    GlobalPruning(v, root.right)
end if
return R
```

**Algorithm 3:** RangeQuery $(start\_key, end\_key, v, root, out\_parR)$

```
$P \leftarrow$ GlobalPruning$(v, r_g)$
foreach $p$ in $P$ do
    if $root$ is not leaf then
        $startc \leftarrow child$ of $root$ whose route directs to $start\_key$ and $v$
        $endc \leftarrow child$ of $root$ whose route directs to $end\_key$ and $v$
        $children \leftarrow$ all children between $startc$ and $endc$
        foreach $node$ in $children$ do
            RangeQuery$(start\_key, end\_key, v, node, R)$
        end foreach
    else if $key$ exists in $root$ then
        add $record$ containing $key$ to $R$
    end if
end foreach
```

As to the implementation of global index in Spark, it is straightforward. We first collect all the partition intervals distributed in the slaves, and then we build a binary search tree as the global index in the master node. The implementation of local indexes in Spark is basically different from the above. One can easily know that RDD is the basic abstraction in Spark, and it represents a partitioned collection of elements that can be operated in parallel. Meanwhile, a partition wraps its dataset records according to its partitioner. Particularly, we observe that RDD is designed for sequential access. This incurs that one cannot build indexes over RDDs *directly*. To deploy the local indexes over RRDs, we use a method suggested in [34]. In brief, we first wrap all the records in a partition into an array with the temporal data format, and construct the local index structure using this array; Afterwards, the local array can be released, and we persist the local in memory to support subsequent queries.

In addition, it would be nice to enable users to write concise SQL statements to support analytics for big temporal data. Yet, in Apache Spark there is the corresponding SQL operations/commands. To this end, we develop new Spark

**Algorithm 4:** TemporalAggregation $(g, v, root)$

```
$P \leftarrow$ GlobalPruning$(v, r_g)$
foreach $p$ in $P$ do
    $child \leftarrow$ child of $root$ which satisfies $v \in child.interval$
    if $child$ is leaf then
        return $child.value$
    else
        return $g(child.value, TemporalAggregation(g, v, child)$
    end if
end foreach
```

SQL operations/commands to support temporal data analytics. Several major changes are as follows.

- We design a new keyword "**VERSION**" to support temporal operations with SQL statements. This new keyword can help us reinterpret the **AS of** subclause in SQL Server, endowing it with the new meaning by modifying the SQL plan in the Spark SQL query engine. Specifically, **FOR VERSION AS OF** *version_number* means specifying a *version_number*, where **VERSION** is just the newly introduced keyword. For instance, users can use the following SQL statements to execute a time travel exact query mentioned in Section 2.

**SELECT** * **FROM** $D$ **WHERE** key = '9'
**FOR VERSION AS OF** $v_2$.

- In order to manage indexes for temporal data, we also develop index management SQL commands. Users can specify the index structure by using **USE** *index_type*, where *index_type* is the keyword for a specific index name (e.g., MVB-TREE, SBTREE). For example, to create a SB-tree index called "sbt" for table $D$, one can use the following SQL commands:

**CREATE INDEX** sbt **ON** $D$ **USE** SBTREE.

# 4 Experiments

In this section we first present the experimental settings (Section 4.1), and then cover and analyze the experimental results (Section 4.2).

## 4.1 Experiment Setup

In our experiments, we use both real and synthetic datasets described as follows. The real dataset **SX-ST** is extracted from a temporal network on the website Stack Overflow [24]. The network has 2.6 million nodes representing users, and 63 million edges in form of $(u, v, t)$, where $u$ and $v$ are the `ids` of source user and of target user respectively, and $t$ is the interaction time between these two users. Specifically, we extract users who interacted with others more than once. And we treat each of these users as a record, in which two consecutive interaction timestamps of the user are regarded as the interval of the record, and the value of the record is the total number of interactions related to the users. This gives us about 0.4 million records. Following the schema of SX-ST, we also generate the synthetic dataset, shorted as **SYN**. Specifically, in SYN the starting timestamp of a record is generated randomly, and the length of the interval is uniformly distributed between the minimum and maximum length of that in SX-ST. The size of SYN ranges from 1 million to 4 billion records (i.e., $[10^6, 4 \times 10^9]$, the default value is $5 \times 10^8$).

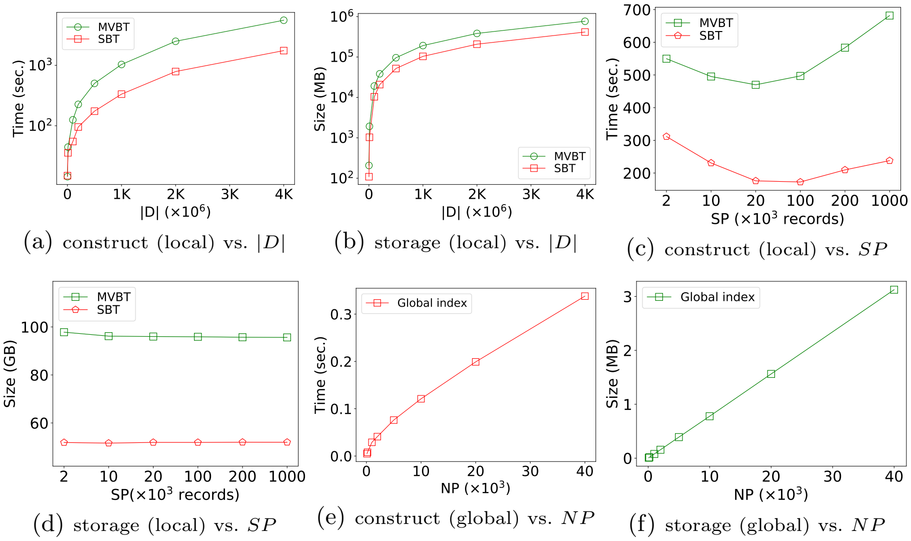
To measure the performance of our system, we adopt two widely-used evaluation metrics: (i) runtime (i.e., query latency) and (ii) throughput. To obtain the runtime, we repeatedly perform 10 queries for each test case, and calculate the average value. On the other hand, the throughput is evaluated as the number of queries performed *per* minute. Additionally, we also examine the performance of indexes used in our system.

We compared our system with two baselines: (i) a Naive In-memory based Solution on Spark (**NISS**). It partitions all temporal records randomly using the default method in Spark, and stores the data in memory of the distributed system. These partitions are collected and managed via RDD, which allows us to manipulate the data in parallel. To achieve temporal queries, NISS uses predicates (e.g., *WHERE* predicate) provided by Spark SQL, to launch a scanning on the data. By checking each record according the condition presented in the query input, NISS can obtain the query result. For example, when an aggregation query with `MAX` function is detected, NISS checks each partition in parallel. For each partition, it scans the whole partition and determines the max value among records, which are alive in version $v$, in this partition. Finally, it collects all "local" max values from partitions and finds the "global" max value. And (ii) a distributed disk-based solution named **OcRT**, which is extended from OceanRT [39]. Note that, OceanRT employs a hashing of temporal data blocks according to the temporal attributes of records; this behaviour essential serves as a global index. In our baseline, we implement this hashing process by grouping the starting value of intervals to form a partition. In addition, OceanRT runs multiple computing units on one physical node and connects these units using Remote Direct Memory Access(RDMA); this behaviour is roughly same to the executors in Apache Spark. More importantly, our adapted solution OcRT stores the data on disks, which is same to that in OceanRT.

All experiments are conducted on a cluster containing 5 nodes with dual 10-core Intel Xeon E5-2630 v4 processors @ 2.20 GHz and 256 GB DDR4 RAM. All these nodes are connected to a Gigabit Ethernet switch, running Linux operating system (Kernel 4.4.0-97) with Hadoop 2.6.5 and Spark 1.6.3. One of these 5 nodes is selected as the master and the remaining 4 machines are *slaves*. The configuration is totally 960 GB main memory and 144 virtual cores in our cluster, which is deployed in standalone mode. In our experiments, the size of HDFS block is 128 MB. The default partition size (a.k.a., the size of each partition) contains $10^5$ records. The balance factor (i.e., fanout) of local index(es) is set to 100.

### 4.2 Experimental Results

Fig. 5 investigates the index cost of our system. For the local indexes, the construction time of SB-Tree (SBT) is much faster than that of MVB-Tree (MVBT), as shown in Fig. 5(a). This is mainly because MVBT requires *node copy* and has about 2 times of operations (e.g.., insertion and deletion) than SB-tree. Even so, the indexing time is acceptable. For example, indexing 4 billion records using MVBT takes about 198GB memory space (cf., Fig. 5(b)), yet it takes only 1.54 hours. Besides, we also show the results by varying the size of partition ($SP$); see Figs. 5(c) and 5(d). It can be seen that there is a non-linear relationship between $SP$ and the index construction time (cf., Fig. 5(c)). This is main because the index construction time is influenced by not only the size of each partition



(a) construct (local) vs. $|D|$ (b) storage (local) vs. $|D|$ (c) construct (local) vs. $SP$

(d) storage (local) vs. $SP$ (e) construct (global) vs. $NP$ (f) storage (global) vs. $NP$

**Fig. 5.** index constrution time and storage overhead vs. $|D|$, $SP$ and $NP$

but also the total number of partitions. In our experiments, the "good" partition size falls in the range from $20K$ to $200K$ records. This is essentially why we choose $SP = 100K$ as the default setting (recall Section 4.1). Note that, an appropriate choice on the number of partitions and the size of each partition can both improve system throughput query latency performance. Meanwhile, we can see that $SP$ makes less impact on the index size (cf., Fig. 5(d)). This further shows that the index size is mainly related to the dataset size $|D|$. On the other hand, one can see that the construction of the global index is very fast; about 330 milliseconds even if $NP$ is set to the largest value (cf., Fig. 5(e)). This is mainly because the global index size is very small, i.e., only about 3 MB even so $NP = 40K$ (cf., Fig. 5(f)). In addition, as we expected, the global index size is strictly proportional to $NP$.

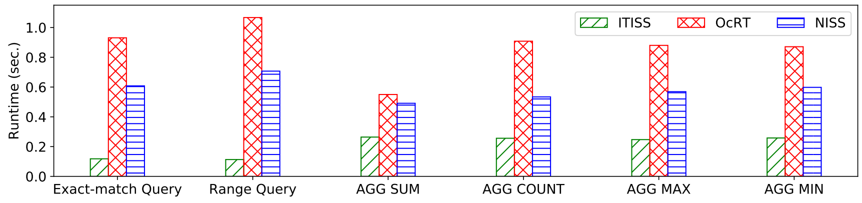
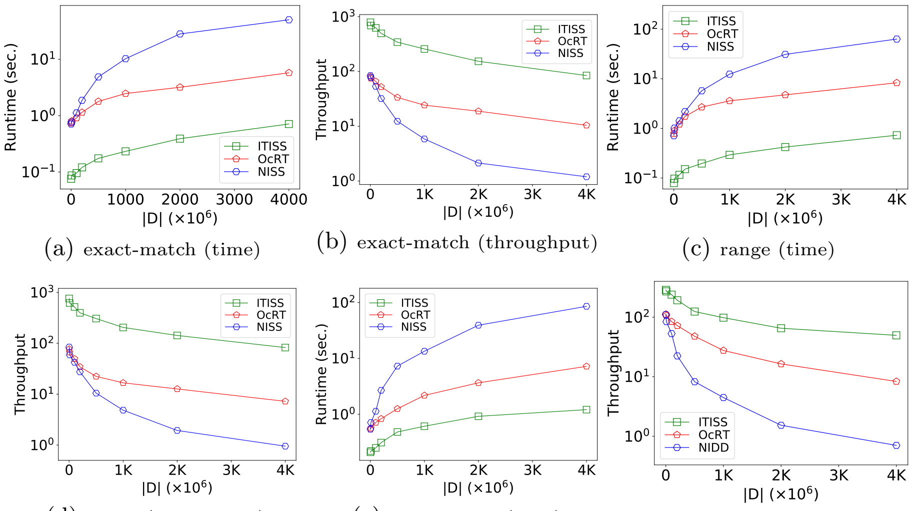


**Fig. 6.** time travel and temporal aggregation queries on the SX-ST dataset

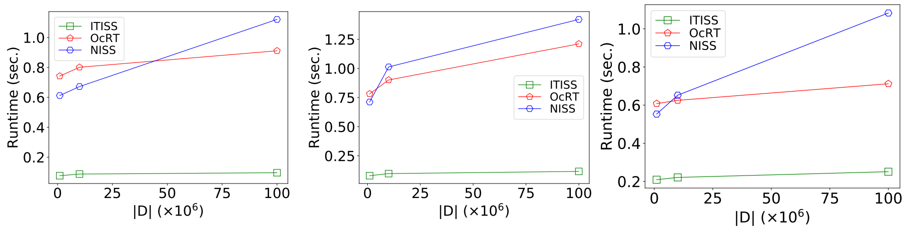
Next, we compare our method with the baselines. We first discuss the results on the SX-ST dataset. It can be seen from Figure 6 that the execution of NISS is



(a) exact-match (time) (b) exact-match (throughput) (c) range (time)

(d) range (throughput) (e) aggregation (time) (f) aggregation (throughput)

**Fig. 7.** Time travel and temporal aggregation queies on the SYN dataset

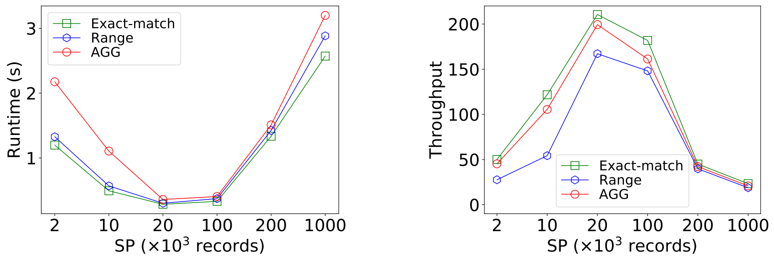


(a) exact-match (time) (b) range (time) (c) aggregation (time)

**Fig. 8.** A enlarged drawing. Here $|D|$ ranges from $1 \times 10^6$ to $10 \times 10^6$

slow, although it also stores the data in-memory. This is mainly because the full scan over the dataset in partitions is time-consuming. As to OcRT, the hashing process can perform partition pruning, but the lack of local index makes it slow, since it needs in-partition full scanning. The reason why OcRT is slower than NISS could be due to two points: (i) OcRT is disk-based solution; and (ii) the partition pruning effect of OcRT is weak when it is confronted with relatively small dataset like SX-ST. Compared to the baselines, our method takes only about 0.3 seconds for temporal aggregation queries, and less than 0.2 seconds for time travel. It is about 3× faster than NISS, and 4× faster than OcRT. This demonstrates the competitiveness of our method. On the other hand, one can see that different aggregation queries (e.g, SUM, MAX) have the similar query cost. In what follows, when we discuss aggregation queries, we mainly report the SUM aggregation query results for saving space.

Fig. 7 covers the comparison results on the synthetic (SYN) data, which is much larger than the SX-ST dataset. For time travel exact-match queries, one



(a) runtime

(b) throughput

**Fig. 9.** Temporal operations vs. $SP$

can easily see from Fig. 7(a) that our solution is 3∼7 times faster than OcRT. Our solution outperforms NISS about one order of magnitude on both *runtime and throughput* (cf., Figs. 7(a) and 7(b)) when dataset size $|D|$ ranges from $10^6$ to $4\times10^9$ records; especially, it outperforms NISS near to two orders of magnitude when $|D| = 10^9$. This essentially demonstrates the superiorities of our solution. Also, we can see that the performance of our framework drops much slower than that of others, which essentially shows us that our framework has much better scalability. This is mainly because the partition pruning in our framework is much more powerful on larger datasets. Another interesting phenomenon is that, OcRT here is obviously better than NISS (cf., Figs. 7(a), 7(c), and 7(e)), while it is inferior than NISS in the previous test (cf., Fig. 6). This is mainly because SX-ST is relatively small, compared to SYN. Fig. 8 well explains this phenomenon (see the crossing point in this figure).

As we expected, when we execute the time travel range queries (cf., Figs. 7(c) and 7(d)), our solution presents the similar performance, compared again the exact-match queries. For example, the running time for both queries is close and has the similar growth tendency). One the other hand, for temporal aggregation queries, one can see from Fig. 7(e) that, the runtime of aggregation query is a little longer than that of time travel operations. This is mainly because it needs to checks many more records. Similarly, in Fig. 7(f), the throughput of the aggregation query has the similar charateristics.

Fig. 9 shows the impact of partition size $SP$ on the performance of temporal queries. we can see from Fig. 9(a) that, the good partition size for both time travel and temporal aggregation queries is between 20K to 100K records. Meanwhile, it can be seen from Fig. 9(b) that the throughput is even more sensitive to partition size. This shows the significance of number of partitions in distributed systems.

## 5 Related Work

In the field of temporal databases, prior works addressed various issues related to temporal data (see several representative surveys [18, 30, 17]).

In the existing literature, most of early works concentrate on semantics of time [6], logical modelling [33] and query languages [4] for temporal data. Recently, some researchers addressed the problem of discovering/mining interesting information [27] from temporal data, such as trend analysis [15] and data clustering [36]. Other works addressed query or search issues for temporal data, such as top-k queries [26] and membership queries [22]. Some optimal problems related to temporal data are also investigated, such as finding optimal splitters for large temporal data [23]. Similar to general databases, in temporal databases join operation is also a common operation, researches on this topic can be found in [13]. Since temporal data is involved with an evolving process, researchers have attempted to model evolutionary traces [32], and to trace various elements in temporal databases, such as tracing evolving subspace clusters [16]. The aforementioned works are related to ours (since these works also handle temporal data). Yet, it is not hard to see that they are clearly different from ours, since our work focuses on time travel and temporal aggregate queries, instead of the above problems such trend analysis, logical modelling.

Nevertheless, there are already existing works addressing the problems of time travel [20, 11, 5, 28, 3, 31, 1] and temporal aggregate [38, 10, 20, 11, 21, 25] queries. For example, Kaufmann *et al.* [20] proposed a unified data structure called timeline index for processing queries on temporal data, in which they use column storage to mange temporal data. General-purpose temporal index structures can be found in [11, 5]. Furthermore, SAP HANA [12] provides a basic form of time travel queries based on restoring a snapshot of a past transaction. ImmortalDB [28] is another system that supports time travel queries. From industry perspective, database vendors, such as Oracle [3], IBM [31], Postgres [1], SQL Server [2], also integrate time travel queries into theirs systems. On the other hand, Snodgrass *et al.* [21] introduced the first algorithm for computing temporal aggregation on constant intervals. Later, algorithm for temporal aggregation based on AVL Trees was proposed [8]. Furthermore, temporal aggregates with range predicates [38], or over extreme cases such as null time intervals [10], are also investigated. Attempts for temporal aggregates with a multiprocessor machine can be found in [25, 19]. Efficient indexing structures supporting temporal aggregates are discussed in [11, 35, 29]. A major feature of the aforementioned proposals or systems is that, they focused on single-machine-based solutions, while few attention has been made on developing distributed solutions for handling big temporal data.

Essentially, we also realize that, distributed temporal analytics for big data have been also investigated in recent years [39, 9]. And they are different from the early work [14] (in which the data being processed is relatively small). Nevertheless, these works share at least two common features: (i) they are distributed disk-based temporal analytics instead of distributed in-memory based temporal analytics; and (ii) time travel and temporal aggregate queries are not covered in their papers. Thus, they are different from our work.

## 6 Conclusion

In this paper we suggested a distributed in-memory analytics framework for big temporal data and implemented it on Spark. Our framework used a two-level index structure to enhance the pruning power. It also provided declarative SQL query interface that enables users to perform typical temporal operations with a few lines of SQL statements. We conducted extensive experiments to demonstrate its superiorities, compared against state-of-the-art solutions. In the future, we plan to extend this framework to support more temporal queries.

## References

1. "Postgres 9.2 highlight - range types." [Online]. Available: http://paquier.xyz/postgresql-2/postgres-9-2-highlight-range-types
2. "Temporal tables." [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables
3. "Workspace manager valid time support." [Online]. Available: https://docs.oracle.com/cd/B28359_01/appdev.111/b28396/long_vt.htm#g1014747
4. I. Ahn and R. Snodgrass, "Performance evaluation of a temporal database management system," in *SIGMOD*, vol. 15, no. 2, 1986, pp. 96–107.
5. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," *VLDB Journal*, vol. 5, no. 4, pp. 264–275, 1996.
6. C. Bettini, X. S. Wang, E. Bertino, and S. Jajodia, "Semantic assumptions and query evaluation in temporal databases," in *SIGMOD*, vol. 24, no. 2, 1995, pp. 257–268.
7. R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas, "R-tree based indexing of now-relative bitemporal data," in *VLDB*, 1998, pp. 345–356.
8. M. H. Böhlen, J. Gamper, and C. S. Jensen, "Multi-dimensional aggregation for temporal data," in *EDBT*, vol. 3896, 2006, pp. 257–275.
9. B. Chandramouli, J. Goldstein, and S. Duan, "Temporal analytics on big data for web advertising," in *ICDE*, 2012, pp. 90–101.
10. K. Cheng, "On computing temporal aggregates over null time intervals," in *DEXA*, 2017, pp. 67–79.
11. R. Elmasri, G. T. Wuu, and Y.-J. Kim, "The time index: An access structure for temporal data," in *VLDB*, 1990, pp. 1–12.
12. F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The sap hana database–an architecture overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.
13. D. Gao, S. Jensen, T. Snodgrass, and D. Soo, "Join operations in temporal databases," *VLDB Journal*, vol. 14, no. 1, pp. 2–29, 2005.
14. J. A. G. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass, "Parallel algorithms for computing temporal aggregates," in *ICDE*, 1999, pp. 418–427.
15. S. Gollapudi and D. Sivakumar, "Framework and algorithms for trend analysis in massive temporal data sets," in *CIKM*, 2004, pp. 168–177.
16. S. Günnemann, H. Kremer, C. Laufkötter, and T. Seidl, "Tracing evolving subspace clusters in temporal climate data," *Data Min. Knowl. Discov.*, vol. 24, no. 2, pp. 387–410, 2012.

17. M. Gupta, J. Gao, C. C. Aggarwal, and J. Han, “Outlier detection for temporal data: A survey,” *TKDE*, vol. 26, no. 9, pp. 2250–2267, 2014.
18. C. S. Jensen and R. T. Snodgrass, “Temporal data management,” *TKDE*, vol. 11, no. 1, pp. 36–44, 1999.
19. M. Kaufmann, P. M. Fischer, N. May, C. Ge, A. K. Goel, and D. Kossmann, “Bi-temporal timeline index: A data structure for processing queries on bi-temporal data,” in *ICDE*, 2015, pp. 471–482.
20. M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May, “Timeline index: A unified data structure for processing queries on temporal data in sap hana,” in *SIGMOD*, 2013, pp. 1173–1184.
21. N. Kline and R. T. Snodgrass, “Computing temporal aggregates,” in *ICDE*, 1995, pp. 222–231.
22. G. Kollios and V. J. Tsotras, “Hashing methods for temporal data,” *TKDE*, vol. 14, no. 4, pp. 902–919, 2002.
23. W. Le, F. Li, Y. Tao, and R. Christensen, “Optimal splitters for temporal and multi-version databases,” in *SIGMOD*, 2013, pp. 109–120.
24. J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, 2014.
25. T. C. Leung and R. R. Muntz, “Temporal query processing and optimization in multiprocessor database machines,” in *VLDB*, 1992, pp. 383–394.
26. F. Li, K. Yi, and W. Le, “Top-k queries on temporal data,” *VLDB Journal*, vol. 19, no. 5, pp. 715–733, 2010.
27. C. Loglisci, M. Ceci, and D. Malerba, “A temporal data mining framework for analyzing longitudinal data,” in *DEXA*, 2011, pp. 97–106.
28. D. Lomet, R. Barga, M. F. Mokbel, and G. Shegalov, “Transaction time support inside a database engine,” in *ICDE*, 2006, pp. 35–35.
29. S. Ramaswamy, “Efficient indexing for constraint and temporal databases,” in *ICDT*, 1997, pp. 419–431.
30. J. F. Roddick and M. Spiliopoulou, “A survey of temporal knowledge discovery paradigms and methods,” *TKDE*, vol. 14, no. 4, pp. 750–767, 2002.
31. C. M. Saracco, M. Nicola, and L. Gandhi, “A matter of time: Temporal data management in db2,” *IBM Corp.*, 2010.
32. P. Wang, P. Zhang, C. Zhou, Z. Li, and H. Yang, “Hierarchical evolving dirichlet processes for modeling nonlinear evolutionary traces in temporal data,” *Data Min. Knowl. Discov.*, vol. 31, no. 1, pp. 32–64, 2017.
33. X. S. Wang, S. Jajodia, and V. Subrahmanian, *Temporal modules: An approach toward federated temporal databases*, 1993, vol. 22, no. 2.
34. D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, “Simba: Efficient in-memory spatial analytics,” in *SIGMOD*, 2016, pp. 1071–1085.
35. J. Yang and J. Widom, “Incremental computation and maintenance of temporal aggregates,” in *ICDE*, 2001, pp. 51–60.
36. Y. Yang and K. Chen, “Temporal data clustering via weighted clustering ensemble with different representations,” *TKDE*, vol. 23, no. 2, pp. 307–320, 2011.
37. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *NSDI*, 2012, pp. 2–2.
38. D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger, “On computing temporal aggregates with range predicates,” *TODS*, vol. 33, no. 2, p. 12, 2008.
39. S. Zhang, Y. Yang, W. Fan, L. Lan, and M. Yuan, “Oceanrt: Real-time analytics over large temporal data,” in *SIGMOD*, 2014, pp. 1099–1102.