Noname manuscript No.
(will be inserted by the editor)

ITISS: An Efficient Framework for Querying Big Temporal Data

Zhongpu Chen[1], Bin Yao[1], Zhi-Jie Wang[2,5,6], Wei Zhang[1], Kai Zheng[3], Panos Kalnis[4], Feilong Tang[1]
chenzhongpu@sjtu.edu.cn, yaobin@cs.sjtu.edu.cn, wangzhij5@mail.sysu.edu.cn, zhangweilst@sjtu.edu.cn, zhengkai@uestc.edu.cn, panos.kalnis@kaust.edu.sa, tang-fl@cs.sjtu.edu.cn

the date of receipt and acceptance should be inserted later

Abstract In the real word, temporal data can be found in many applications, and it is rapidly increasing nowadays. It is urgently important and challenging to manage and operate big temporal data efficiently and effectively, due to the large volume of big temporal data and the real-time response requirement. Processing big temporal data using a distributed system is a desired choice, since a single-machine based system usually has the limited computing ability. Nevertheless, existing distributed systems or methods either are disk-based solutions, or cannot support native queries, which may not well meet the demands of low latency and high throughput. To attack these issues, this article suggests a new approach to handle big temporal data. Our approach is an In-memory based Two-level Index Solution in Spark, dubbed as ITISS. The proposed framework of our solution is easily understood and implemented, but without loss of effectiveness and efficiency. Based on the proposed framework, this article develops targeted algorithms for handling time travel, temporal aggregation, and temporal join queries, respectively. We have implemented our framework in Apache Spark, extended the Apache Spark SQL to support declarative SQL interface that enables users to perform temporal queries with a few lines of SQL statements, and conducted extensive experiments to verify the performance of our solution. The experimental results, based on both real and synthetic datasets, consistently demonstrate that our proposed solution is efficient and competitive for processing big temporal data.

Keywords temporal join query · time travel query · temporal aggregation query · distributed in-memory systems · big data

1 Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China.
2 School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China.
3 School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China.
4 King Abdullah University of Science and Technology, Mecca, Saudi Arabia.
5 Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, China
6 National Engineering Laboratory for Big Data Analysis and Applications, Beijing, China.

1 Introduction

Temporal data management is a hot topic in the filed of databases, and it has been extensively studied in the past decades. Recently, it is attracting more and more attention [23, 33, 15, 25], owing to its wide applications.

– For instance, users may want to investigate the demographic information of an administrative region (e.g., Texas) at a specific time (e.g., three years ago). Querying a historical version of the database (like mentioned above) is usually known as time travel [7, 16, 36, 39, 26].
– Consider another example, in the quality assurance (QA) department users may want to examine and analyze how many orders are delayed as a function of time, thereby querying all historical versions of the database over a certain time period (e.g., from March 1, 2017 to December 31, 2017). Queries like mentioned above are usually referred to as temporal aggregation [25, 15, 26].
– As for temporal join, it is also usually required to obtain results by combining two or more temporal datasets [17, 68, 40]. For example, given two datasets which recording foraging behaviors of two kinds of wild animals in a region respectively, zoologists may be interested in how many pairs of these two kinds of animals forage in overlapping periods.

In the past many years, there are already a large number of works addressing the problems of time travel, temporal aggregation and temporal join queries (see e.g., [40, 7, 26, 16, 27, 32, 68, 36, 18]). There are also some similar heuristic researches in similarity search and matching [48, 47, 64, 65, 63, 49], trajectories search [45, 51, 50] and path planning in spatial networks [46, 52, 53] with proposed approaches. Nevertheless, in the literature most of existing works focused on studying single-machine based solutions, and only few effort has been made on developing distributed solutions for processing big temporal data.

Nowadays, various types of applications, e.g., web apps, mobile apps, and Internet of things (IoT) apps, generate more and more temporal data, and the volume of generated temporal data is increasingly large. It is very important and urgently needed to effectively and efficiently handle big temporal data and keep the useful information as much as possible. In the meanwhile, it is also challenging to process such a large volume of temporal data in traditional database systems, since the limited computing and storage ability of a single-machine based system. For example, DiDi [1], a leading mobile transportation platform, collects more than 70TB trajectories with timestamp information, which can hardly be handled in a single commodity machine.

It is clear that dealing with such a large volume of temporal data using a distributed system should be a good choice. In recent several years, distributed analytics for big temporal data have been also studied (see e.g., [69, 12]). Overall, these works share two common features at least:

– (i) They are distributed *disk-based* analytics for big temporal data.
– (ii) Time travel, temporal aggregation and temporal join queries are not covered in their papers.

With the surging data volume, these methods/solutions could not well satisfy the requirements of low latency and high throughput. On the other hand, Spark SQL

[1] https://www.didiglobal.com/

[66] is such an engine that extends Spark (a fast distributed *in-memory* computing engine) to enable users to query the data with the SQL interface inside Spark programs. Nevertheless, currently, it cannot provide native support for temporal operations such as time travel, temporal aggregation and temporal join.

To support distributed in-memory analytics for big temporal data with low latency and high throughput, this article presents an In-memory based Two-level Index Solution in Spark, dubbed as ITISS. To the best of our knowledge, none of existing big data systems (such as Apache Hadoop, Apache Spark) provide native support for querying big temporal data, and none of previous works develop distributed in-memory based solution for handling time travel, temporal aggregation and temporal join over big temporal data. In summary, the main contributions of this article are as follows:

- It presents a distributed in-memory analytics framework for querying big temporal data. The proposed framework is easily understood and implemented, but without loss of efficiency.
- It develops targeted algorithms for addressing time travel, temporal aggregation and temporal join queries, by fully utilizing the suggested framework that utilizes a two-level index structure.
- It implements the proposed framework in Apache Spark, and extends the Apache Spark SQL to support declarative SQL interface that enables us to perform temporal queries with a few lines of SQL statements.
- It conducts a comprehensive empirical study for the proposed solution, based on both real and synthetic temporal data. Extensive experimental results consistently demonstrate that the proposed solution is efficient and also competitive compared against the competitors adapted from state-of-the-art techniques.

Roadmap. The rest of this paper is structured as follows. We review prior works that are most related to ours in Section 2. Particularly, we point out that this work is a valued-added version of our previous study. We present some preliminaries and define our problems formally in Section 3. In Section 4, we present the proposed framework for processing big temporal data, including a distributed indexing structure, the detailed query algorithms. The implementation details based on Apache Spark are discussed in Section 5. In Section 6, we present a comprehensive empirical experimental evaluation. Finally, we conclude this article in Section 7.

2 Related Work

In the domain of temporal databases, previous works [23, 24, 42] addressed variety of issues related to temporal data. In what follows, we review prior works by classifying them into three categories. Firstly, we discuss previous works related to temporal data processing, and then we review prior works related to time travel, temporal aggregation, and temporal join queries. Finally, we review prior works related to the distributed disk-based temporal analystics.

2.1 Temporal Data Processing

In the past decades, most of early works focused on logical modelling [56], semantics of time [8], and query languages [4] for temporal data. For example, Ahn and Snodgrass [4] proposed a prototype of a temporal database management system that was built by extending Ingres. Their system supports the temporal query language TQuel, a superset of Quel, handling four types of database static, rollback, historical and temporal. Bettini *et al.* [8] studied two types of semantic assumptions: point-based and interval-based. Wang *et al.* [56] introduced a concept, called a temporal module, to resolve differences or mismatches among the constituents.

Recently, some researchers addressed the problem of discovering/mining interesting information from temporal data. For example, Gollapudi and Sivakumar [20] proposed a framework based on relational records and metric spaces to study trend analysis for massive temporal data. Yang and Chen [61] studied temporal data clustering via weighted clustering ensemble with different representations. Loglisci *et al.* [35] developed a temporal data mining framework for analyzing longitudinal data. There are also some other works that addressed query or search issues for temporal data. For example, Li *et al.* [33] proposed a data structure named seb-tree to support top-k queries for temporal data. Kollios and Tsotras [28] studied membership queries and presented a more general problem of temporal hashing. Temporal data can also be combined with other types of data (e.g, spatial data) [14,59,70] and thus it can extend the classical queries [13,11,34, 71]. Besides, some optimal problems related to temporal data are also investigated, such as finding optimal splitters for large temporal data [30,55,22].

The works mentioned above are related to ours, because these works also process temporal data. Nevertheless, it is not hard to understand that they are obviously different from ours, because our work focuses on time travel, temporal join, and temporal aggregation queries, instead of the above mentioned problems such as trend analysis, finding optimal splitters, and logical modeling.

2.2 Time Travel, Temporal Aggregation and Temporal Join

We also realize that there are already a lot of existing works discussing the problems of time travel [3,7,26,16,36,43,1] , temporal aggregation [15,67,26,16,27,32] and temporal join [68,26,69,18] queries.

For example, Elmasri *et al.* [16] studied the general-purpose temporal index structures. Their methods can be used to retrieve versions of objects that are valid during a specific time period, and support the processing of the temporal WHEN operator and temporal aggregate functions efficiently. Becker *et al.* [7] developed a multi-version B-tree that supports insertions and deletions of data items at the current version. In addition, their methods can achieve range queries and exact match queries for any version, current or past. Kaufmann *et al.* [26] proposed a unified data structure called timeline index for processing queries on temporal data, in which they use column storage to mange temporal data. In addition, SAP HANA [17] gives a basic form of time travel queries, based on the idea of restoring a snapshot of a past transaction. Compared to SAP HANA, ImmortalDB [36] is another system that supports time travel queries, and particularly it builds transaction time database support into a database engine, not in middleware. From the perspective of industry, database vendors, such as Postgres [1], Oracle [3], IBM [43], and SQL Server [2], also integrated time travel queries into theirs systems.

On the other hand, Kline *et al.* [27] introduced the first algorithm for evaluating temporal aggregation queries on constant intervals. After that, Böhlen *et al.* [10] presented the algorithm for computing temporal aggregation queries based on AVL Trees. Furthermore, temporal aggregation queries with range predicates [67], or over extreme cases such as null time intervals [15], were also studied. Some efforts for temporal aggregation queries with a multiprocessor machine were made in [32, 26]. Besides, efficient indexing structures supporting temporal aggregation queries were developed in [60,16,41].

There also exists an abundance of studies for temporal join, and various variants related to temporal join operations and corresponding algorithms can be found in a comprehensive survey [18]. More specifically, temporal join processing with different indexes such as Timeline Index [26], Time Index [54], R*-Tree [68], MVBTree, and B+-Tree was also well studied. In addition, Gunadhi and Segev [21] investigated the variant operation of temporal join, called the temporal intersection join. Spatio-Temporal join operation was also addressed (see e.g., [38]). Furthermore, Lu *et al.* [37] discussed spatially partitioned temporal join. Segev and Gunadhi [44] addressed event-join optimization in temporal relational databases.

A common feature of the aforementioned proposals or systems is that, they focused more of their attention on single-machine-based solutions, while less attention has been made on investigating distributed solutions for processing big temporal data.

2.3 Distributed Temporal Analytics

Essentially, there are some papers that are related to distributed analytics for big temporal data. In other words, distributed temporal analytics for big data have been also investigated in recent years [69,12], and they are different from the early work (e.g., [19]), in which the temporal data being processed is relatively small.

For example, Chandramouli *et al.* [12] studied temporal analytics on big data for Web Advertising, and proposed a framework called TiMR, that combines a time-oriented data processing system with a M-R framework. Zhang *et al.* [69] developed a cloud-based infrastructure over large-scale temporal data such as call logs from a telecommunication company.

Nevertheless, these works share two common features at least: (i) time travel, temporal aggregation and temporal join operations are not mentioned in their papers; and (ii) their works are distributed disk-based temporal analytics, instead of distributed in-memory based temporal analytics. In addition, some systems [29, 5,6,57] focus on big spatio-temporal data. [29,5,6] is based on Hadoop rather than Spark, and therefore it cannot leverage the in-memory advantages. Although [57] is built on Spark, it does not consider *time travel* or *aggregation* operations. More importantly, it uses a spatial-first partition strategy and does not support neither index and SQL.

Finally, there are another lines of on-self general big data systems (e.g., Apache Kafka, Apache Storm, Apache Samza and Apache Kinesis). However, they cannot meet our requirements for big temporal data for many reasons. For example, these systems are designed for streaming (or messaging) data but this property is not our focus, and in-memory advantages are not well exploited. Of course, it is also possible to build stream-oriented big data analytics to meet other requirements based on these these systems.

This paper is a major-value added version of our previous study [62]. In the preliminary word [62], we address distributed in-memory processing for time travel and temporal aggregation queries, and report part of experimental results. The new contributions include (1) We report more experimental results which are not shown in [62]; (2) We further introduce the temporal join operations under ITISS framework; (3) We add the mathematical formulation for the minimal overlap problem for load balance; (4) We give a more comprehensive review on prior works, and strengthen the presentation.

3 Problem Formulation

In this paper, we attempt to achieve three most representative temporal operations (i.e., the time travel queries, temporal aggregation queries, and temporal join queries) over big temporal data in distributed environments. Nevertheless, the proposed framework and algorithms discussed later can be easily extended to support other operations and other data such as *bitemporal data* [9]). Next, we present the formal definitions for the problems to be studied. For ease of reference, we summarize the frequently used symbols/notations in Table 1.

Table 1 Frequently Used Symbols

Notation	Description
D	a temporal dataset
t_i	the i-th temporal record of D
I_p	a partition interval
Q_e	time travel exact-match query
Q_r	time travel range query
Q_a	temporal aggregation query
Q_j	temporal equal range join query
g	a temporal aggregation operator, e.g. *SUM*, *MAX*

We use D to denote a temporal dataset that contains $|D|$ temporal records $\{t_1, t_2, ..., t_{|D|}\}$. Each record t_i is a quadruple in the form of $(key, value, start, end)$, where $i \in [1, |D|]$, *key* refers to the *id* of the record, *start* and *end* correspond to the starting timestamp and the ending timestamp of a time interval, during which the record is currently alive. Further, given a record t_i and a version (or timestamp) v, we say that the record t_i exists in the version v (i.e., the record t_i is alive in the version v), if and only if $v \in [t_i.start, t_i.end)$. Besides, the *end* can be omitted when the record is alive now and its terminating time (in the future) is unknown. Now, consider the third example in Section 1, and let us see how a temporal record is represented in such format. If there is a wild wolf whose id is 006 (assigned by zoologists), and it usually hunts from 10:00am to 12:00pm in grassland A (assigned by zoologists). Therefore, this activity can be recorded as (006, A, 10:00, 12:00). It is worth noting that since timestamps here are discrete and monotonically increasing, they can be abstracted as version ids, and similar

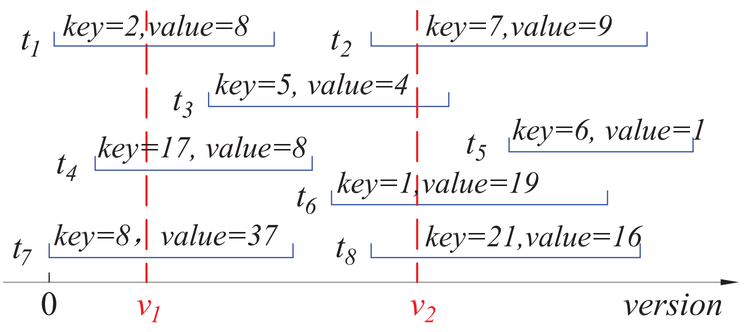


Fig. 1 Illustration of temporal queries.

abstractions are also adopted in [26]. Hence, in the following, "timestamp" and "version" are interchangeably used if the context is clear.

Typically, Time travel is one of the most significant temporal operations in temporal databases, as it builds a consistent view for the history of a database. In this article, we discuss two extensively used time travel operations. That is, *time travel exact-match query* and *time travel range query*. Both of two operations can support querying the past version of a database. Their main difference is that, as for *key* the input of an *exact-match query* adopts a specific value, whereas the input of a *range query* adopts a given range [7,39]. Note that sometimes *key* is not a simple identifier, and specifying ranges for keys is meaningful. For example, the product codes within the range may come from the same assembly line. Formally, their definitions are formulated as follows.

Definition 1 (Time travel exact-match query) Given a time travel exact-match query $Q_e = (key, v)$, it retrieves the record (denoted by R) from the temporal dataset D such that,

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

To understand, consider a simple temporal database containing 8 temporal records as an example in Figure 1. In this case, if one issues a query $Q_e = (17, v_1)$, then the query shall return t_4. Correspondingly, if one issues a query $Q_e = (17, v_2)$, then the query shall return nothing.

Definition 2 (Time travel range query) Given a time travel range query $Q_r = (start\_key, end\_key, v)$, it shall retrieve a set of records (donated by R) from the temporal dataset D such that,

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

Also consider the example shown in Figure 1. Assume that one issues a query $Q_r = (5, 20, v_1)$, then the query shall return $\{t_4, t_7\}$ as the answer. In contrast, if one issues a query $Q_r = (5, 20, v_2)$, then the query shall return $\{t_2, t_3\}$ as the result.

Similar to the time travel operations, Temporal aggregation is also a common operation in temporal database, and it usually is much more challenging and expensive. Temporal aggregation has been heavily investigated, since it was first introduced in [27]. In this article, we focus more of attention on aggregation (e.g., `SUM`, `MIN`, `MAX`) conducted at a specific timestamp. Here, the result is an aggregate value on *values* of temporal records. Specifically, the temporal aggregation operation is formally defined as follows.

Definition 3 (Temporal aggregation query) Given a temporal dataset D and a temporal aggregation query $Q_a = (g, v)$, where g is an aggregation operator such as `SUM`, `MAX`, the query shall return an aggregate value (denoted by R) from the temporal dataset D such that,

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

Again, consider the example shown in Figure 1. Assume that one issues an aggregation query $Q_a = (MAX, v_1)$, then the query shall return 37 (since $max\{8, 8, 37\} = 37$). In contrast, if one issues an aggregation query $Q_a = (SUM, v_2)$, then the query shall return 48 (since $9 + 4 + 19 + 16 = 48$).

Temporal join is another constantly used yet costly temporal operation. Just like queries mentioned above, this query involves both temporal and key domains. As for temporal domain, two tuples are considered to be joined candidates when their time intervals are overlapped; as for key domain, any meaningful predicate is possible. [40] provides several common predicates of temporal join. In this paper, we focus on the temporal domain and users may issue a join to specify the predicate (donated by θ) with respect to the key domain. Formally, it is formulated as follows.

Definition 4 (Temporal join query) Given two temporal datasets D_1, D_2 and a predicate θ in terms of key, a temporal join query is to return pairs of records (donated by R) such that

$$R = \{(t_i \in D_1, t_j \in D_2) \mid \theta(t_i.key, t_j.key) \wedge [t_i.start, t_i.end) \cup [t_j.start, t_j.end) \neq \emptyset\},$$

where $[t_i.start, t_i.end) \cup [t_j.start, t_j.end) \neq \emptyset$ means that there is an overlap between the time interval of t_i and that of t_j.

As an example, consider two temporal datasets $D_1 = \{t_1 = (3, 8, 1, 8), t_2 = (4, 9, 4, 7), t_3 = (5, 7, 2, 5)\}$ and $D_2 = \{t_4 = (3, 10, 3, 9), t_5 = (4, 10, 8, 12), t_6 = (5, 9, 4, 6)\}$. If the specified θ is equality on keys, then the query returns $\{(t_1, t_4), (t_3, t_6)\}$.

Remark 1. Notice that, it is possible to design plenty of queries in terms of time interval and key/value in temporal databases, but we cannot cover them thoroughly in one single paper. To this end, we only select some interesting and well-studied operations. Also, our focus is how to handle big temporal data in distributed environments, and design algorithms based on a general and flexible framework. In the meanwhile, it is worth noting that our framework has its limitations in the patterns of queries, and we will elaborate it after the introduction of its general query processing (i.e., Section 4.2).

As introduced in Section 1, a straightforward implementation based on existing distributed systems/algorithms is inefficient and ineffective. In the next section, we introduce our method in detail.

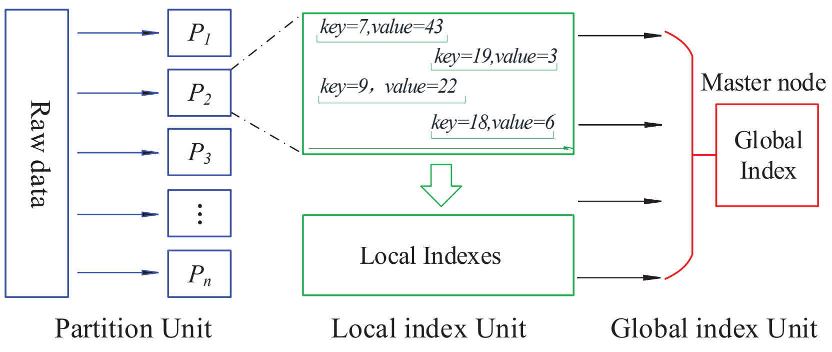


Fig. 2 An illustration of our system framework.

4 The Proposed Method

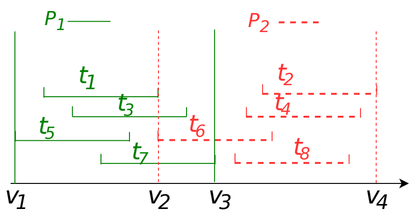
In this section, we first present the distributed system framework. Then, we cover the detailed algorithms for achieving time travel, temporal aggregation and temporal join queries based on the suggested system framework, respectively.

4.1 The Architecture of Our System

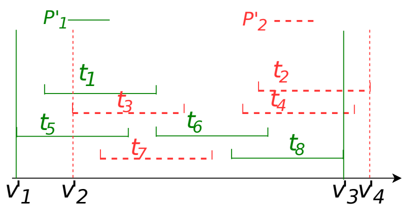
At a high level, the framework of our proposed method is comprised of three main parts:

1. The partition unit. It is responsible for partitioning all the temporal data into distributed (slave) nodes. Normally, in order to keep the *load balance*, the partition method should guarantee that — each node has roughly the same size of data.
2. The local index unit. For each partition, the local indexes are maintained, in order to avoid a "full" scanning. This shall help us boost the query efficiency. Furthermore, in each partition we also maintain a *partition interval* (explained later), which shall be used for the construction of the global index.
3. The global index unit. A global index located in the master node is designed to prune "unpromising" partitions in advance. This design can avoid checking each (individual) partition, and so it shall help us reduce the network transmission and/or cost CPU cost. In our implementation, the master node shall collect all partition intervals from each (individual) partition in the slave nodes, and then build the global index, based on the partition intervals collected before.

The overall architecture of our framework is illustrated in Figure 2. One can easily understand that the proposed framework adopts a two-level indexing structure, which can avoid visiting irrelevant candidates, such as partitions and local records, as much as possible. Though the rationale behind our framework seems to be simple, it is definitely efficient as demonstrated later. In the remainder of this section, we discuss important issues in each of these units.



(a) Range partition method



(b) Another partition method

Fig. 3 An illustration of different partition methods.

4.1.1 Partition Method

It is well known that *load balance* is a desired goal when one partitions the general data, and the main concern is usually to achieve roughly same size for each partition. One standard strategy is to use hash-based method in big data settings, but it is unable to maintain the "balance" when querying big temporal data. As for the temporal data, another important and also desired goal is to minimize the overlap of the partition intervals since this property can maintain the locality of temporal data. To understand, let us compare the two partitioning strategies for 8 temporal records as shown in Figure 3(a). Assume that we want to partition 8 temporal records into 2 partitions. It is clear that the interval overlap between P_1 (containing t_1, t_3, t_5 and t_7) and P_2 (containing t_2, t_4, t_6 and t_8) is much smaller than the one between P'_1 (containing t_1, t_5, t_6 and t_8) and P'_2 (containing t_2, t_3, t_4, and t_7). Consider there is a query whose start and end are v_1 and v_2 respectively, we can safely discard P_2, but we have to keep both P'_1 and P'_2. In other words, the smaller interval overlap, the better the pruning ability. In summary, in order to gain load balance for distributed temporal data when partitioning, we need to achieve both *roughly equal size* of each partition and *the smallest interval overlap* between partitions.

Next, we formulate the smallest interval overlap as an optimization problem.

Definition 5 (Minimum interval overlap problem) A set $D = \{t_1, t_2, \cdots, t_n\}$ is divided into m partitions (i.e., subsets) and the partition is denoted as P_i ($1 \leq i \leq m$) such that the intersection of any different two partitions is empty. Our goal is to find these partitions such that

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} f(P_i, P_j) \\ \text{subject to} \quad & P_i \cap P_j = \emptyset, i \neq j, \\ & P_1 \cup P_2 \cup \cdots \cup P_m = D, \\ & |P_i| \approx \frac{|D|}{m}, i = 1, 2, \cdots, m, \end{aligned}$$

where $f(P_i, P_j)$ is the degree of interval overlap between P_i and P_j. Let $P_i.start$ and $P_i.end$ be the leftmost and rightmost endpoints in P_i respectively; and $P_j.start$ and $P_j.end$ have the similar definitions in terms of P_j. We have

$$f(P_i, P_j) = \max\{0, \min\{P_i.end, P_j.end\} - \max\{P_i.start, P_j.start\}\}$$

To achieve the above two goals, in our design we partition the temporal data by intervals. For clarity, we dub it as *range partition*. A greedy partitioning algorithm is summarized in Algorithm 1. Without loss of generality and for simplicity, we assume that $|D|$ can be divided evenly by m, and for any given record, it is not completely overlapped by a different record. Firstly, we sort the records in D by their interval tuples (i.e., $(start, end)$) in ascending order. Then, in order to balance the size of each partition, every m records are grouped into one partition. Note that in Algorithm 1, $D[i, j]$ denotes the partition where records are indexed from i to j (both are inclusive) in D.

Algorithm 1: RangePartition (D, m)

```
1 Let $P = \emptyset$
2 sort the records in $D$ by intervals in ascending order
3 $k \leftarrow \frac{|D|}{m}$
4 for $i \leftarrow 1$ to $m$ do
5 |   $P_i \leftarrow D[(i-1)k+1, ik]$
6 end for
7 return $P$
```

As an example, we can sort these temporal records in Figure 3(a) by their intervals, and we obtain the sorted records $\{t_5, t_1, t_7, t_3, t_6, t_8, t_4, t_2\}$. After that, we can split evenly these sorted records into two parts. As a result, P_1 shall contain first four records $\{t_5, t_1, t_7, t_3\}$; correspondingly, P_2 shall contain the other four records $\{t_6, t_8, t_4, t_2\}$. In this way, the partition interval of P_1 is $[v_1, v_3)$, and that of P_2 is $[v_2, v_4)$. In this case, the interval overlap of P_1 and P_2 is $v_3 - v_2$, which is the minimum overlap.

Next, we prove that if the number of partitions is 2, the greedy is optimal by contradiction.

Theorem 1 *If $m = 2$, then Algorithm 1 provides an optimal solution to minimum interval overlap problem when there are no any two records completely overlapping each other.*

Proof Let $P_1, P_2, \cdots, P_m$ be the results returned by Algorithm 1. Assume that there is a better solution which is obtained by moving (or exchanging) records between adjacent partitions.

Since there are no any two records completely overlapping each other, when $i < j$, $f(P_i, P_j)$ can be simplified as

$$f(P_i, P_j) = \max\{0, P_i.end - P_j.start\}$$

If $P_i.end \le P_j.start$, then it is obvious that there is no any other methods that can do better. Hence, the assumption is wrong.

Otherwise, the overlap degree is $P_i.end - P_j.start$. In this case, any moving (or exchanging) would cause the increase of overlap degree. In other words, the new solution is not better than previous partitions. Hence, assumption is also wrong.

Since we have a contradiction in both cases, Algorithm 1 provides an optimal solution to minimum interval overlap problem. $\square$

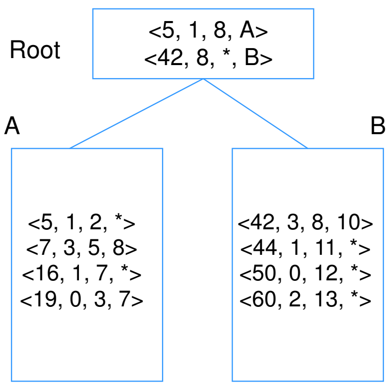


Fig. 4 An illustration of MVB-Tree used in our system.

Further, when we try to increase m, it is reasonable to verify the feasibility of this algorithm, since we can recursively split the partitions (suppose that m is a power of 2).

Remark 2. Algorithm 1 is based on the assumption that there are no any two records completely overlapping each other, so this greedy algorithm fails to work in a general setting. As for the records which completely overlap others, we provide a heuristic method: (1) At first, we filter these records which completely overlap others; (2) Then, the greedy algorithm is conducted on the remaining records; (3) Finally, given a filtered record, we assign it to a partition such that there is a smallest overlap between the filtered one and the partition interval. At the same time, we also shall make a trade-off to keep the number of each partition roughly equal.

4.1.2 Local Index Method

As discussed earlier, the local index serves as managing the temporal data in each partition. In the existing literature, there are already some on-shelf index structures that can support time travel and temporal join queries, e.g., *multi-version B-tree* [7], and *time-index* [16]. In this article, we adopt multiversion B-tree (shorted as MVB-Tree) as an example. For ease of understanding, Fig 4 illustrates this index structure. For non-leaf node, the entry is in the format of $\langle router, start, end, reference\rangle$, where the *router* is a separator key of its children, *reference* is the pointer to its child, *start* and *end* are the minimum and maximum version id of records in its child respectively; *router*, together with *start* and *end*, guides the search for a record. In the leaf nodes, each entry essentially denotes a record, and its format (i.e., $\langle key, value, start, end\rangle$) is the same with the one introduced in Section 3. Note that, in this figure, the symbol $*$ means that this record is still alive currently. In brief, the 1st entry in the root node points to its leaf child A, which includes all the records that are alive from version 1 to 8 (excluded).

Meanwhile, there are already existing index structures (e.g., [60, 41]) that can support temporal aggregation queries. Here we adopt the index (named the SB-Tree, which incorporates segment-tree and B-tree) presented in [60] as an example. In brief, the SB-Tree node consists of two arrays, as shown in Fig 5. More specifi-

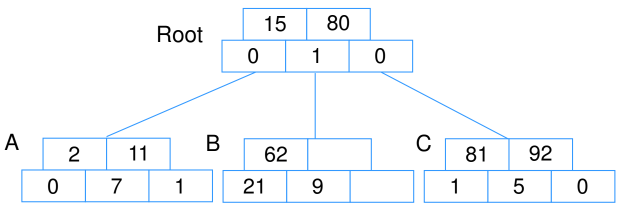


Fig. 5 An illustration of SB-Tree used in our system.

cally, one of the arrays stores and manages the intervals, which is used for pointing to its children nodes, and another stores the values to aggregate. For example, the second time interval of B is $[62, 80)$ and its corresponding value is 9 (note that *key* is omitted in this case). When one wants to calculate an aggregation using the SB-Tree, he/she can search the tree from root to leaf, and aggregate the values in this path.

Note that, although this paper adopts the SB-Tree and MVB-Tree, it is by no mean compulsory to use these indexes. In other words, other on-shelf indexes and/or more excellent indexes developed in the future can also be used in our proposed framework.

4.1.3 Global Index Method

As mentioned previously, the global index serves as managing the partition intervals. Since the partition interval is essentially a pair of version numbers, and is comparable through the starting value and length of the interval; naturally, one can use the well-known binary search tree to manage/maintain these partitions' interval information. Notice that, for each partition in slave nodes, there exist many *time intervals (of records)*. Nevertheless, we only need to use one *partition interval* for a partition. To understand the partition interval, let's consider a simple example. Assume that there are three time intervals $\{[t_1, t_2), [t_3, t_4), [t_5, t_6)\}$ in a partition. Then, the partition interval shall be $[min\{t_1, t_3, t_5\}, max\{t_2, t_4, t_6\})$. This way, each partition interval in the global index essentially corresponds to a specific partition in slave nodes. This means that, in the query processing, if a partition interval can be pruned, the corresponding partition can be safely pruned. Based on this observation, in our design each node in the global index shall maintain a key-value pair $< id, I_p >$, where id and I_p are the partition id and its corresponding temporal interval, respectively.

4.2 Query Processing

The query evaluation in our framework is comprised of two stages: (i) the global pruning, and (ii) the local look-up.

– Phase 1: global pruning. Essentially, the first phase is to fully exploit the version v (in the query input) and the global index to prune "unrelated" partitions. To understand, let us consider an example shown in Figure 6. Assume that one wants to prune the partitions that do not belong to version 53, she/he can

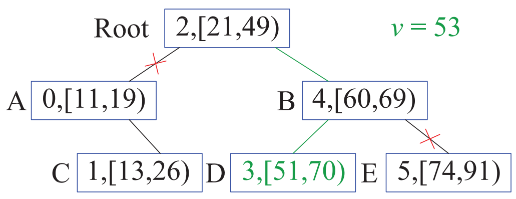


Fig. 6 An illustration of global pruning.

examine the partition interval by traversing the global index. As a result, only one partition whose ($id = 3$) can be acted as the candidate.
- Phase 2: local look-up. Based on the local indexes and part of query inputs, the second phase mainly retrieves, in each of candidate partitions, the "qualified" records. To understand, consider the example in Figure 4 again. Now, assume that there is a time travel exact-match query $Q_e = (5, 7)$, then the local look-up would be guided to A, and B is filtered. Finally, this procedure will return $\langle 5, 1, 2, * \rangle$ as result after checking the entries in A.

As we can see, the conciseness of query processing in ITISS is simultaneously reassuring and disappointing. It is reassuring because its conciseness makes it easy to implement and understand. However, it is also disappointing because its conciseness would cause some limitations in patterns of queries. To be specific, ITISS works only when the queries can be easily divided into the two stages as mentioned above. On the other hand, although our framework is able to achieve relatively low latency (as shown in experimental results), it is mainly designed for batch-processing generally and it fails to handle streaming data due to the different data patterns and computing techniques, and we will leave it as an open problem in our future work.

In the remainder of this section, we present the detailed query algorithms for time travel, temporal aggregation and temporal join queries, respectively.

4.2.1 Time Travel Queries

In this part, we first present algorithms for the time travel exact-match query, and then explain how to achieve the time travel range query.

The pseudo-codes of the *time travel exact-match query* are detailed in Algorithm 2. Note that, Line 2 is used to perform the global pruning; the details of this function are illustrated in Algorithm 3. In Algorithm 2, after finishing the global pruning at the master node, we can obtain the `ids` of candidate partitions, which are stored in P. Then, the local look-up (Lines 3-11) searches the results in each of candidate partitions. Notice that, here the local look-ups for all these candidate partitions are distributed to the cluster and executed in parallel.

A running example. It is given in the overall stages mentioned at the beginning of Section 4.2.

As for the *time travel range query*, the detailed procedure is similar to that of Algorithm 2. The major difference is that, we do not need to search the *child* for the given key. Instead, we maintain an array of *children* that can direct to $[start\_key, end\_key]$, and then we check each node in *children*.

Algorithm 2: ExactMatchQuery (key, v)

```
1  Let $R = \emptyset$
2  $P \leftarrow$ GlobalPruning($v, r_g$)        // $r_g$ is the root of the global index
3  foreach $p$ in $P$ do
4      $node \leftarrow r_l$                      // $r_l$ is the root of the local index
5      while $node$ is not a leaf do
6          $node \leftarrow child$ of $node$ whose route directs to $key$ and $v$
7      end while
8      foreach $record$ in $node$ do
9          if $record.key = key$ then
10             put $record$ into $R$
11         end if
12     end foreach
13 end foreach
14 return $R$
```

Algorithm 3: GlobalPruning $(v,\ root)$

```
1  Let $P = \varnothing$
2  if $root \neq null$ then
3      if $v \in root.I_p$ then
4          add $root.id$ into $P$
5      end if
6      GlobalPruning($v$, $root.left$)
7      GlobalPruning($v$, $root.right$)
8  end if
9  return $P$
```

4.2.2 Temporal Aggregation Queries

When we process the temporal aggregation queries, we also employ the global pruning at first. The pruning process is the same with that for dealing with the time travel queries. Yet, the local look-up process works in a different manner.

Algorithm 4: LocalAggregation $(g, v, root)$

```
1  child $\leftarrow$ root.child which satisfies $v \in [child.start, child.end)$
2  if child is leaf then
3      record $\leftarrow$ child.entry which satisfies $v \in [record.start, record.end)$
4      return child.value
5  else
6      return g(child.value, LocalAggregatation(g, v, child))
7  end if
```

Briefly speaking, in each of candidate partitions, we design a recursive algorithm to conduct the local aggregation. This search process starts from the root node of the candidate partition, until the node matching this condition is found. If the found one is a leaf node, then we just need to return the aggregate value in it. Otherwise, we shall recursively find the aggregate value of its child whose interval contains v. The pseudo-codes for local temporal aggregation queries are illustrated in Algorithm 4. It is worth noting that: (1) At most one child of a node can satisfy the condition that contains v (Lines 1 and 3); (2) For some function (e.g., *AVERAGE*), we may store a pair of $(SUM, COUNT)$ for incremental updates. In this case, we shall make a slight changes to Algorithm 4 correspondingly. Again, the local aggregations are conducted in parallel and their results would be collected to master node.

A running example. After the global pruning we can get some candidate partitions. Consider the example in Figure 5, and suppose it is the index structure of one of the candidates. Given an aggregation query where $v = 64$ and $g = SUM$. At first, we find that the second interval of *root* meets the condition (i.e., $15 < 64 < 80$), and we further explore the tree into B recursively. Then, we find that the second interval of B also meets the condition (i.e., $62 < 64 < 80$). Thus, we aggregate values (i.e., $1 + 9 = 10$) in the path from *root* to B.

4.2.3 Temporal Join Queries

Unlike time travel and temporal aggregation queries, temporal join query is involved with two datasets. Without index support (e.g., Spark SQL), it is necessary to compute a Cartesian product from the two datasets, and then filters based on join predicates are followed. Now, we are able to avoid the time-consuming Cartesian product generating process by leveraging the two-level index in ITISS. Our join algorithm consists of global and local join.

Global join. To boost the performance, it is important to set a proper partition size to determine the number of join partitions (see Section 6.2). We firstly use the global index to generate pairs of partitions candidates to join. To be specific, given two partitions P_i and P_j from D_1, D_2 respectively, (P_i, P_j) can be candidate pair only if their time intervals are overlapped. Thus, candidate partitions pairs are generated following this rule and then they are sent to slaves for parallel processing.

Local join. Given a candidate pair (P_i, P_j), we perform a local join. Given the index of P_i, for each record in P_j, we perform a procedure like Algorithm 2 to return pairs of records. In fact, if the join predicate θ is the equality with respect to *keys*, then we can reuse the procedure of time travel exact-match query. Since for different θ, the resulting local join algorithms differ each other, we do not provide the pseudo codes here. It is worth noting that the local join here only takes the index of one partition, and it is possible for us to design an algorithm to use dual indexes, but in fact, one-index will not lose the efficiency in practice and it is easier to implement in the framework. We also conducted experiments to verify this design choice in Figure 21.

A running example. Given a candidate pair (P_i, P_j) after the global filtering, assume that the index structure of P_j is the one in Figure 4. Suppose one of records in P_j is $t = (6, 2, 3, 7)$. If the join predicate θ is the gap between keys of two records is less than 2. By leveraging the index, the searching process would be guided to A, and the set of pairs $\{(\langle 5, 1, 2, *\rangle, t), (\langle 7, 3, 5, 8\rangle, t)\}$ will be returned.

Remark 3. We can find that the global pruning in algorithms mentioned above only takes the information of partitions' intervals and it is not aware of the local index. In other words, the global pruning in ITISS still works even if the local indexes are not built in advance as long as the information of partitions' intervals is given. In general, generating the interval information is lightweight task compared with creating index. In this case, we shall build the local index on the fly. Therefore,

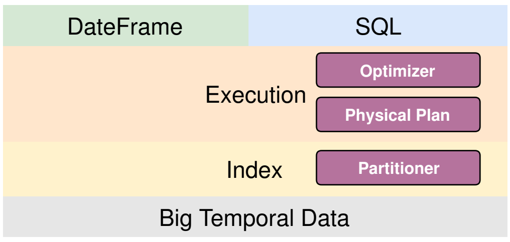


Fig. 7 Structure of system implementation.

we can make a trade-off between heavy indexing process over whole data and relatively long query time in practice. For example, when the data is changing sometimes, we may prefer to build the local index on the fly if the underlying local index does not support incremental updates.

5 Implementation on Apache Spark

In this section, we discuss how to implement our proposed framework in Apache Spark.

As illustrated in Figure 7, for the big temporal data, we read it as RDD. After that, we create index over RDD, and the key technique is to design a load-balanced partitioner. Our execution is based on the two-level index, and we shall rewrite the physical plan and temporal-aware optimizer. At last, we provide both DataFrame and SQL interface API to end users.

5.1 Core Implementation

To support the partition method mentioned in Section 4.1, we extend Spark's RangePartitioner. Notice that, Spark's RangePartitioner is developed for the general purpose data partition; it cannot support *partition by interval* effectively. To achieve this, we implement the comparison procedure for the interval data format, and integrate it to Spark RangePartitioner.

Regarding the implementation of global index in Spark, we first collect all the partition intervals distributed in the slave nodes, and then we construct a binary search tree (BST) as the global index in the master node.

Yet, the implementation of the local indexes in Spark is basically different from the strategy mentioned above. To understand, it is helpful to mention the RDD (Resilient Distributed Dataset) in Apache Spark. As we know, RDD is fault-tolerant and can be stored in memory to support fast data reusing, without the need of accessing disk. Moreover, it is the basic abstraction in Spark, and it represents a partitioned collection of elements, which can be handled in parallel. In the meantime, a partition wraps the dataset records, according to its partitioner. In particular, we realize that RDD is designed for sequential access. This incurs that we cannot *directly* build indexes over RDDs. In order to deploy the local indexes over RRDs, we adopt a method suggested in [58]. Briefly speaking, we first load all the temporal records (in a partition) into the memory, and then we build the local index structures. When it comes to implementation, we also need to decouple the index and raw data in order to make our system more flexible. Finally, we persist the local index structures in memory, so as to support subsequent queries. As mentioned in Remark 3, the design trade-off also needs to be considered. In practice, we may resort to building on the fly strategy if the data changes relatively frequently.

5.2 Extension to Spark SQL

Furthermore, it would be nice to enable users to input concise SQL statements to conduct analytics for big temporal data. But there is no corresponding SQL commands in Apache Spark. To address this issue, we design new Spark SQL commands/operations to support analytics for big temporal data. The main changes are listed as follows.

– To efficiently manage and operate indexes for temporal data, we design the index management SQL statement. This way, users can specify the index structure by using **USE** *index_type*, where *index_type* refers to the keyword for a specific index name (e.g., SBTREE, MVBTREE). To understand, consider an example: assume that we want to create an SB-tree index called "sbt" for table D, then we can use the following SQL statement:

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

– Furthermore, to support temporal operations with SQL statements, we also design a novel keyword "**VERSION**". This new keyword can help us reinterpret the **AS OF** sub-clause inherited from SQL Server. This way, we can endow it with the new meaning by revising the SQL plan in the Spark SQL engine. More specifically, **FOR VERSION AS OF** *version_number* means specifying a *version_number*, where **VERSION** is just the newly introduced keyword. For example, assume that one wants to execute the time travel exact-match query, temporal aggregation query, and temporal join query mentioned in Section 3, he/she can use the following SQL statements, respectively.

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

  SELECT SUM(value) **FROM** D **WHERE** key = 12
  FOR VERSION AS OF v_2.

  SELECT * **FROM** D_1 **JOIN** D_2 **ON** $D_1.key = D_2.key$.

6 Performance Evaluation

In this section, we first describe the experimental settings including datasets, compared methodologies, evaluation metrics, parameter settings and the experimental platform (Section 6.1). Then, we report and analyze our experimental results (Sections 6.2 $\sim$ 6.6).

6.1 Experimental Settings

In our experimental evaluation, we use both real and synthetic datasets detailed as follows. Since the size of synthetic dataset is larger, it can show the potential when processing big temporal data. Thereby, we will use it by default.

– The real dataset called SX-ST, which is extracted from a temporal network on the website "Stack Overflow" [31]. The temporal network has 2.6×10^6 nodes that represent users, and 63×10^6 edges in form of (u, v, t), where v and u denote the ids of target and source users respectively, while t is the interaction time between these two users. More specifically, we extract from the network the users who interacted with other users more than once. Particularly, we view each of these users as a record, in which two consecutive interaction timestamps of a user are looked as the interval of the corresponding record, and the value of the record is represented by the total number of interactions related to the users. In summary, this gives us about 0.44×10^6 records.
– We also generate the synthetic dataset, dubbed as SYN, by following the schema of SX-ST. Specifically speaking, in SYN dataset the starting timestamp of a record is randomly generated, while the length of the interval of the correspond record is distributed uniformly between the minimum and maximum length of that in the SX-ST dataset. The size of the SYN dataset ranges from 1×10^6 records to 4×10^9 records for time travel and temporal aggregation operations. These records take disk space from 32MB to 166GB. In our experiments, the default setting for the number of records is 5×10^8 records. As for the temporal join operation, the left dataset and right dataset are of the same size, which ranges from 1 million to 10 million (i.e., $[10^6, 10^7]$ records in the SYN dataset. In our experiments, the default value for the number of records for temporal join operation is 2×10^6, unless stated otherwise.

Following most of prior works that addressed distributed query processing, in our experiments we also use two widely-used evaluation metrics, in order to measure the performance and efficiency of our proposed system:

– The running time (i.e., the query latency). To obtain the running time, we perform repeatedly 10 queries for each test case, and then report the average value of the time spent on these queries to avoid randomness.
– The throughput. It refers to the number of queries performed *per* minute. Usually, the throughput is inversely proportional to the running time.

In addition, we also test the performance of indexes used in our system. Note that, although we directly use existing indexing techniques, it is still meaningful to discuss its construction time and storage cost. This is because the experimental results would show the feasibility of these techniques in a distributed setting.

For ease of examining the efficiency and competitiveness of our proposed solution, we compare our solution/system with two baselines adapted from state-of-the-art techniques:

– A distributed disk-based solution called `OcRT`. This solution is extended from OceanRT in [69]. Notice that, OceanRT runs multiple computing units on one physical node, and it connects these units by using the Remote Direct Memory Access (RDMA); roughly, this behaviour is the same as the executors in Apache Spark. Furthermore, OceanRT adopts a hashing of temporal data blocks based on the temporal attributes of the corresponding records; this behaviour serves essentially as a global index. As for this baseline, we implement this hashing process by grouping the starting value of intervals to form a partition. More importantly, the adapted solution, OcRT, stores the temporal data on disks, which is the same with that in OceanRT.
– A Naive In-memory based Solution on Spark (`NISS`). It randomly partitions all temporal records by using the default approach in Spark, and stores the temporal data in memory of the distributed system. These partitions are collected and managed through Resilient Distributed Dataset (RDD), which allows users to manipulate the managed data in parallel. To perform temporal queries, the baseline, NISS, utilizes the predicates (e.g., *WHERE* predicate) provided by Spark SQL, to launch a scanning on the data. NISS can finally obtain the query result, by checking each record according to the condition contained in the query input. For instance, when a temporal aggregation query with the `MAX` operator is detected, the baseline NISS shall check each partition in parallel. As for each partition, NISS shall scan the whole partition and find out the "max" value of all the records that are alive in version v. At last, it shall collect all "local" max values from these partitions and find out the "global" max value. Consider another example, the temporal join query is implemented by the nested loop method in each partition. To be specific, for each record of the first operand in a partition, we examine all the records of the second operand to see whether it is a "join" candidate. Then, the "local" join results shall be collected and form the final answer.

All of our experiments are performed on a cluster that contains 5 nodes with dual 10-core Intel Xeon E5-2630 v4 processors @ 2.20 GHz and 256 GB DDR4 RAM. All these five nodes are connected to a Gigabit Ethernet switch, running Linux operating system (Kernel 4.4.0-97) with Spark 1.6.3 and Hadoop 2.6.5. One of these five nodes is chosen as the *master*, and the remaining four machines are as *slaves*. Overall, the configuration is of 144 virtual cores and 960 GB main memory in the cluster, which is deployed in the standalone mode. Furthermore, in the experiments the default partition size (i.e., the size of each partition) contains 1.0×10^5 records. The size of Hadoop Distributed File System (HDFS) block is 128 MB, and the fanout (i.e., the branch number) of the local index(es) is set to 100, unless stated otherwise. At last, the default values in our experiments are summarized in Table 2.

6.2 The Impact of Partition Size

It is important and also interesting to study the impact of the partition size. Here we vary the partition size SP and to check how the performance changes. We first

Table 2 Default values in experiments

Item	Value
data size for time travel and temporal aggregation	5×10^8
data size for temporal join	2×10^6
partition size	1×10^5
branch factor	100

discuss the results of time travel and temporal aggregation queries, and then report the results of temporal join. This is because the former two queries have similar complexity while the join is much slower in general. More importantly, the partition size would have more impact on join queries than others as it would generate $O(N \times N)$ candidate partitions (blocks), where N is the number of partitions.

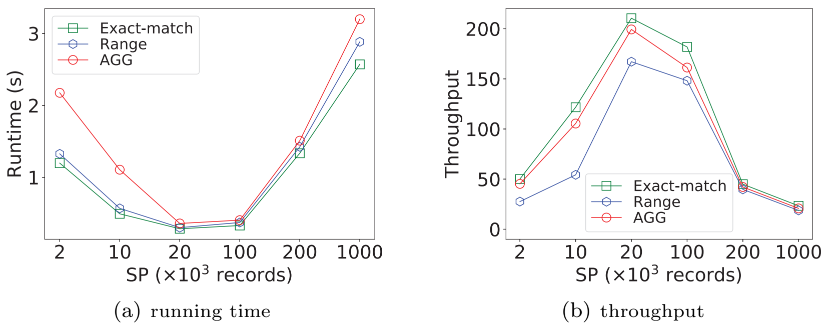


Fig. 8 The performance of time travel and temporal aggregation queries vs. SP.

► *Time travel and temporal aggregation.* Figure 8 shows the experimental results of time travel (including the exact-match query and the range query) and temporal aggregation operations. It can be seen from Figure 8(a) that, the good partition size for both the time travel queries and the temporal aggregation queries is between 20K and 100K records. This essentially tells us that it could be better to set the partition size to a value in the range of $[10 \times 10^3, 100 \times 10^3]$, which can assure the best performance for the proposed method ITISS.

► *Temporal join.* Given the size of dataset, the number of partitions (NP) and the partition size (SP) actually provide the same semantics. To see the impact of partition size, we take ITISS as an example and vary the NP in the range of $1, 2, 4, 8, 16, 32$.

The optimal number of partitions for the temporal join operation is around 4, and its corresponding size of partition is 500K, as shown in Figure 9. This is mainly because, for the ITISS method, the temporal join query generally requires smaller number of partitions to eliminate the cost incurred by the data copying process. This further shows us that it is important to choose the number of partitions in distributed systems.

In summary, this experiment can guide us to choose a "good" partition size for our system, and it also implies the underlying reason to choose the default partition size in our experimental setting.

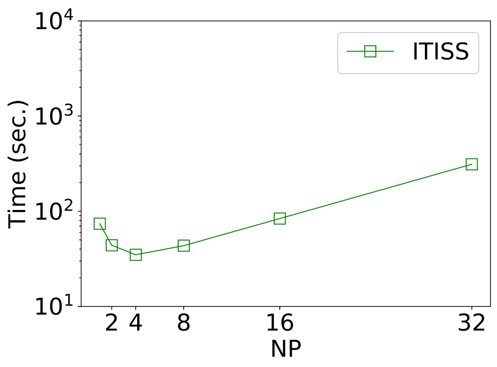


Fig. 9 The performance when varying NP.

6.3 The Results of Index Cost

In this subsection, we first investigate the construction and storage cost of local indexes, and then the ones of the global index.

▶ *Local indexes.* As for the local indexes, it can be seen from Figure 10(a) that, the construction time of MVB-Tree (MVBT) is much longer than that of SB-Tree (SBT). The main reason is that, the MVB-Tree needs to perform the node copy, and the operations such as insertion and deletion are about two times than that of the SB-Tree, which naturally consumes much more time in the construction process. Nevertheless, the overall indexing construction time is still acceptable. For instance, it takes only 1.54 hours for indexing 4 billion records by using the MVB-Tree. In addition, Figure 10(b) shows the indexing storage overhead for both the MVB-Tree and SB-Tree. As we expected, the storage overhead grows when the dataset size increases.

▶ *Varying the size of partition.* Besides, Figure 11(a) shows the results by varying the size of partition. One can easily see that there is a non-linear relationship between the index construction time and the size of partition (i.e., SP). The main reason is that, the index construction time is decided by not only the total number

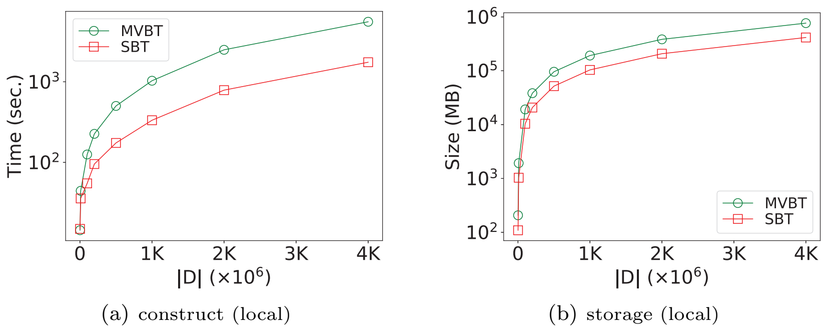


Fig. 10 Index construction time and storage overhead vs. $|D|$.

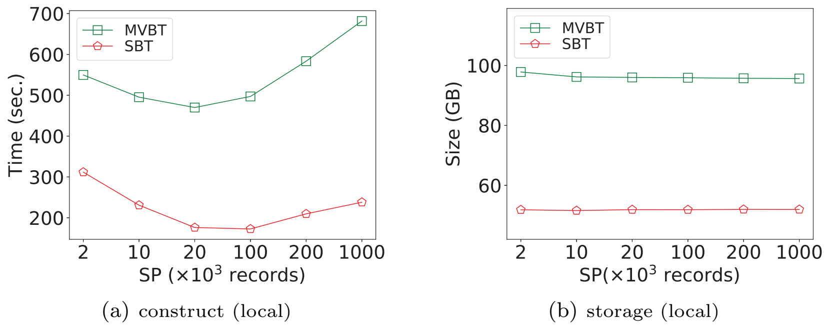


Fig. 11 Index construction time and storage overhead vs. SP.

of partitions but also the size of each partition. From this figure, it can be seen that the "good" partition size falls in the range from $20{\times}10^3$ to $200{\times}10^3$ records. This essentially explains why we choose $10{\times}10^3$ as the default setting of the partition size (recall Section 6.1). It is worth noting that an appropriate choice on the size of each partition and the number of partitions can both improve the system performance (i.e., obtaining a higher throughput and a lower query latency). In addition, Figure 11(b) shows the storage cost when we vary the size of partition. It is easily see that the partition size makes less impact on the storage cost of the index (a.k.a., the index size). This set of results further validate that the index size is mainly relevant to the size of dataset (i.e., $|D|$).

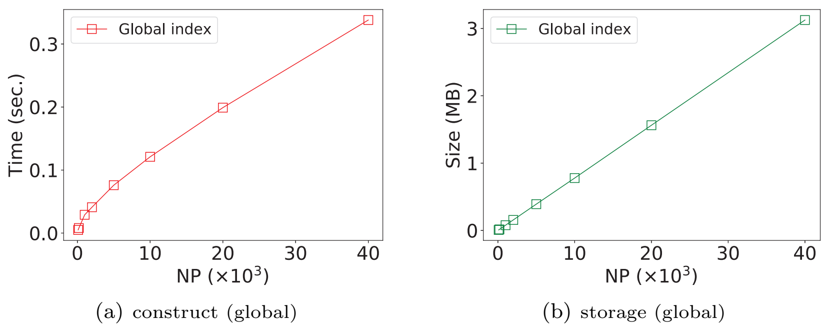


Fig. 12 Index constrution time and storage overhead vs. NP.

▶ *Global index.* Figure 12(a) shows the construction cost of the global index. It can be seen that the construction process is very fast. It takes about only 330 milliseconds even when the number of partitions is set to the largest value (i.e., $40{\times}10^3$). The main reason is that, the global index size is very small. For example, when the number of partitions is equal to $40{\times}10^3$, the index size is only about 3 MB, as shown in Figure 12(b). On the other hand, as we expected, it can be seen from Figure 12(b) that, the size of global index is strictly proportional to the number of partitions.

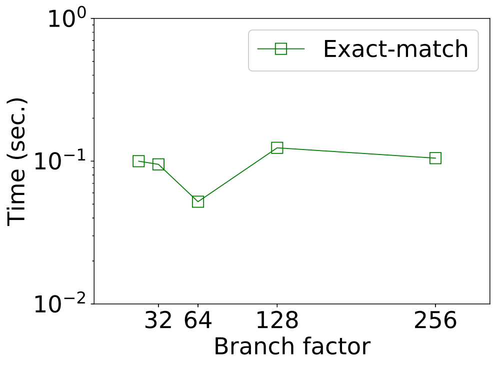


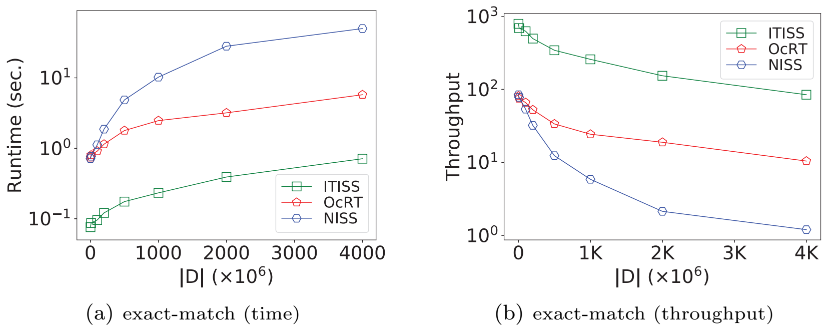
Fig. 13 The performance when varying branch factor.

▶ *Varying index parameters.* Now we take the time travel exact-match query as an example to investigate the branch factor's impact on performance. Figure shows the results when we vary it from 16 to 256.

We can find that it performs best when the branch factor is around 64, and this is mainly caused by tradeoff between depth of a tree and branch factor for a better pruning ability. Hence, we choose 100 as the default branch factor in experimental setting.

6.4 Compared Results on The Synthetic Dataset

In this subsection, we investigate temporal operations on the synthetic dataset. Similar to the previous subsection, we first investigate the time travel queries (including exact-match query and range query), and then temporal aggregation and temporal join queries.



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

Fig. 14 Results of time travel exact-match queies on the synthetic dataset.

▶ *Time travel exact-match queries on SYN.* Figure 14 covers the comparison results of time travel exact-match queries on the synthetic (SYN) data, which is much

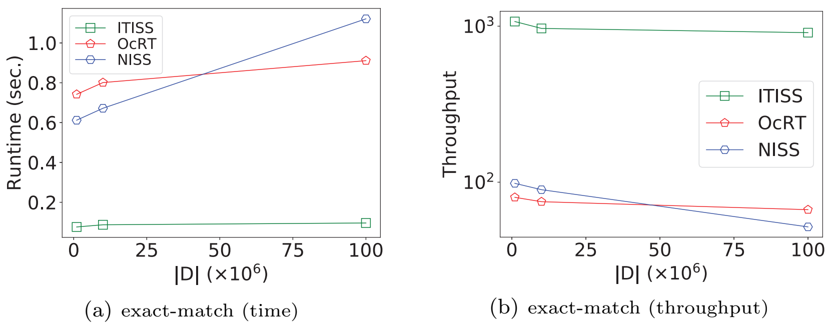


Fig. 15 The more detailed results for time travel exact-match queries on the synthetic dataset. Here $|D|$ ranges from 1×10^6 to 100×10^6.

larger than the SX-ST dataset. For the time travel exact-match queries, we can easily see from Figure 14(a) that our proposed method ITISS is 3∼7 times faster than the OcRT method. In addition, our proposed method outperforms the NISS method about one order of magnitude on both *runtime and throughput* when the dataset size $|D|$ ranges from 1×10^6 to 4×10^9 records (cf., Figure 14(a) and 14(b)); especially, it outperforms the NISS method near to two orders of magnitude when $|D|$ is equal to 4×10^9. This essentially validates the superiorities of our proposed solution. Moreover, we can see also that the performance of our proposed solution drops much slower than that of others, which essentially demonstrates that our proposed solution has much better scalability. The main reason is that, the partition pruning in our framework is much more powerful on larger datasets.

Another interesting observation is that, the OcRT method here seems to be better than the NISS method (cf., Figures 14(a) and 14(b)), while it is inferior to the NISS method in the previous test (cf., Figure 22). This is mainly because the SX-ST dataset is relatively small, compared to the SYN dataset. Figure 15 well explains this interesting phenomenon; see the crossing point between the red and blue lines in the corresponding sub-figures such as Figure 15(a).

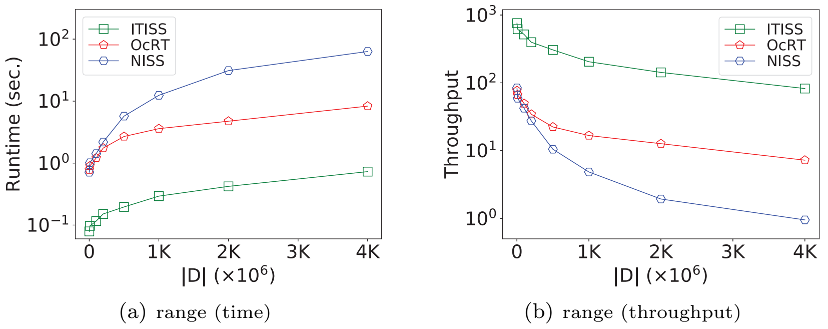
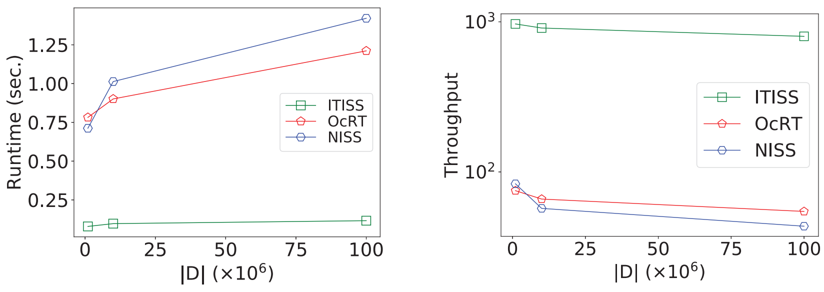


Fig. 16 Results of time travel range queies on the synthetic dataset.

► *Time travel range queries on SYN.* As we expected, compared with the time travel exact-match queries, our proposed solution exhibits the similar performance for the time travel range queries (see Figures 16(a) and 16(b)). For instance, the running time for both time travel exact-match query and time travel range query is close, and has the similar change tendency. Moreover, we can also see from Figure 16 that, here the OcRT method seems to better than the NISS method while it is inferior to the NISS method in Figure 22. The reason is the same as our previous explanations. That is because, the real dataset SX-TX (cf., Figure 22) is small, compared against the synthetic dataset. Figure 17 further explains the phenomenon.

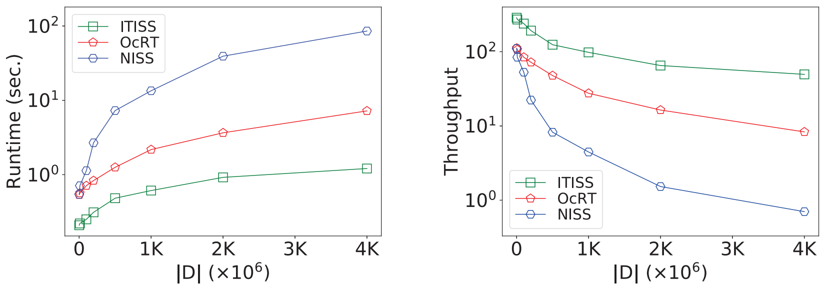


(a) range (time) (b) range (throughput)

Fig. 17 The more detailed results for time travel range queries. Here the dataset size $|D|$ ranges from 1×10^6 to 100×10^6.

► *Temporal aggregation queries on SYN.* Figure 18(a) reports the results of temporal aggregation queries. On one hand, by comparing 18(a) with Figures 14 and 16, we can see that the runtime for the aggregation query is a little longer than that of time travel operations (e.g., exact-match query, and range query). The main reason is that the temporal aggregation query needs to examine many more records.

As for the throughput (cf., Figure 18(b)), it has the similar behaviours. That is, the throughput for the temporal aggregation operations is a little smaller than that of time travel operations. Also, we observe from Figure 18 that the OcRT method



(a) aggregation (time) (b) aggregation (throughput)

Fig. 18 Results of temporal aggregation queies on the synthetic dataset.

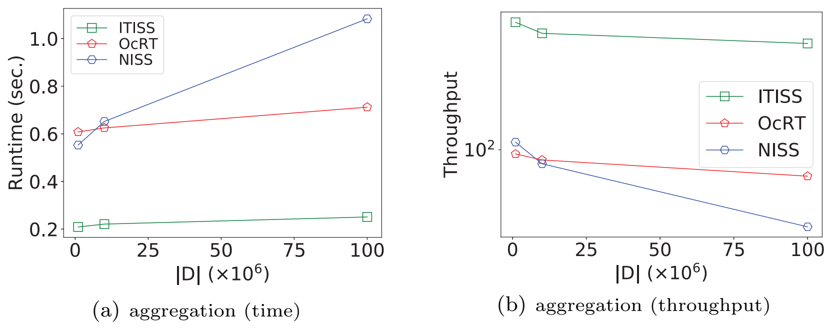


Fig. 19 A more detailed results. Here $|D|$ ranges from 1×10^6 to 100×10^6.

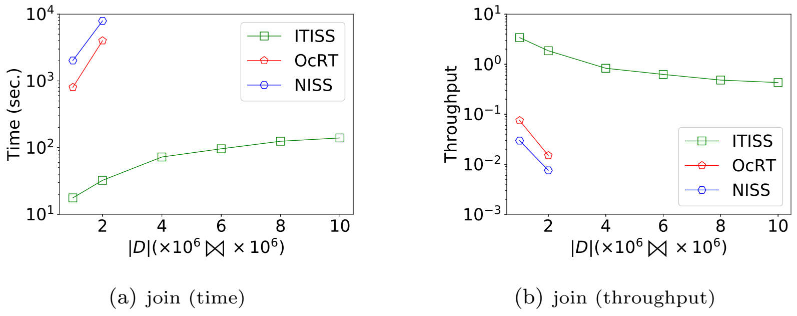


Fig. 20 TER-Join query on the SYN dataset.

seemingly exhibits the better performance, compared to the NISS method. The more details shown in Figure 19 clarify the underlying reason.

► *Temporal join queries on SYN.* In what follows, we report the experimental results of temporal join operation on the SYN dataset.

Figure 20 reports the experimental results of the proposed method ITISS for the temporal join queries. As we expected, the runtime increases when the data size increases. On the other hand, we can see that, compared with the time travel and temporal aggregation queries, the runtime here is larger and the throughput is smaller. This is mainly because the temporal join query needs to operate on two datasets (i.e., the left dataset and the right dataset, recall Section 4.2.3), and checks the matching relation. Nevertheless, as shown later, our proposed method ITISS has larger throughput and shorter runtime, compared against the competitors. As similar as our proposed method ITISS, the runtime curve of OcRT increases and the throughput curve drops when the dataset size increases. In addition, although we use this much larger dataset, it can be seen from the results that, our proposed method ITISS is more than ten times faster than the OcRT method when the dataset size is 10×10^6. This essentially demonstrates the superiorities of our method. As for NISS method, when the size of dataset is more than 2×10^6 records, NISS costs more than 10 hours computation time, so we terminate the program manually. On the other hand, it is easily see that, when the size of dataset is below 6×10^6 records, ITISS is about 2 orders of magnitude faster than NISS,

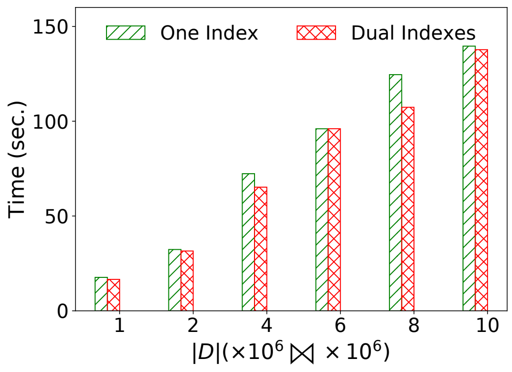


Fig. 21 The comparison resluts of two indexing techniques of local join.

and the runtime increases much slower as the size of dataset grows. This further illustrates the effectiveness and scalability of our proposed method.

Next, we evaluate the performance differences of two different indexing strategies for local join. The first one (donated by *OneIndex*) is to build an index for one candidate partition and traverse the records in another one (as described in Section 4.2.3). The second choice (donated by *DualIndexes*) is to build two indexes for the candidate pairs of partitions respectively, and execute a local join on these two indexes. Figure 21 reports the results. The interesting finding is that the running time of two different indexing techniques is similar. In other words, dual indexes cannot lead to a direct performance improvement when designing the temporal join algorithm, and this is mainly because the pruning ability is not better when applying *Dual Indexes*. Thus, in practice, we would prefer the *One Index*, since it is easier to implement and we are able to save index building time if we have to build it in the fly.

6.5 Compared Results on The Real Dataset

In this part of experiments, all the results are obtained based on the SX-ST datasets, and we only report the running time. We first discuss the results of time travel (here we use two typical time travel query versions: exact-match query, and range query), and then discuss the results of temporal aggregation (here, we use four aggregation query versions: agg sum, agg count, agg max, and agg min), and finally we discuss the results of temporal join.

► *Time travel queries on the SX-ST dataset.* It can be seen from Figure 22 that the execution of NISS is slow on the time travel operations including both exact-match query and range query, although this method also stores the data in-memory. The main reason is that, it needs to perform the full scan over the dataset in partitions, which is time-consuming. As for the method OcRT, the hashing process can achieve partition pruning, which is benefit to the efficiency. Yet, this method lacks the local index, it requires to perform in-partition full scanning, which makes this method slow. The reader could be curious why the OcRT method is slower

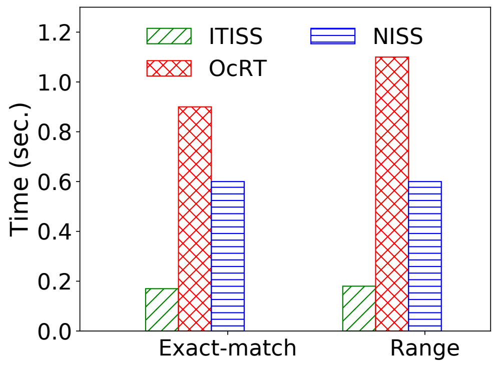


Fig. 22 The comparison resluts of time travel queries on the SX-ST dataset.

than the NISS method. The reasons are twofold. Firstly, the OcRT is disk-based solution. Secondly, the partition pruning force of the OcRT method is poor when the dataset needing to be processed is relatively small (e.g., the SX-ST dataset). On the other hand, we can observe that, compared against the baselines, our method takes less than 0.2 seconds for time travel operations including exact-match query and range query. Roughly, it is 9× faster than the OcRT method, and 5× faster than the NISS method. This essentially demonstrates the competitiveness of our proposed method.

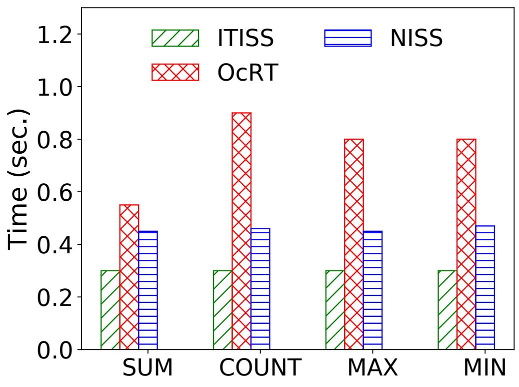


Fig. 23 The comparison resluts of temporal aggregation queries on the SX-ST dataset.

► *Temporal aggregation queries on the SX-ST dataset.* Figure 23 reports the comparison results of temporal aggregation operations. Similar to time travel operations discussed before, here the ITISS still exhibits the strongest performance among these tree methods. Specifically, our method takes only about 0.3 seconds for temporal aggregation queries. It is about 2× faster than NISS, and 2× ∼ 4× faster than OcRT. This further 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.

► *Temporal join queries on the SX-ST dataset.* Now we investigate the TER-Join operation on the real dataset SX-ST. In the experiments, the size of both of the datasets to be joined (i.e., left and right datasets) is 0.44×10^6 records. Note that we only have one real word dataset, so we generate another by adjusting the intervals randomly.

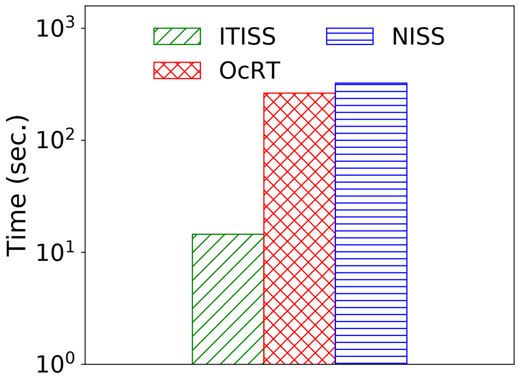


Fig. 24 The comparison resluts of temporal join queries on the SX-ST dataset.

From Figure 24, we can observe the huge performance gap between our proposed ITISS and OcRT (NISS). It is obvious that we can get larger performance (more than ×10) boosting in temporal join than that in time travel or aggregation due to the larger complexity of join queries.

6.6 Summary of Experimental Results

As for the local index, there is a non-linear relationship between partition size and the index construction time. Nevertheless, the overall construction time of the used indexes is acceptable. As for the global index, the index size is strictly proportional to the number of partitions, and its construction is very fast, about 330 milliseconds even if the number of partitions is set to the largest value. In addition, the partition size has the significant impact on the query performance, and so it is important to choose the number of partitions in distributed systems.

On the real dataset, the proposed method takes about 0.3 seconds for temporal aggregation query, and takes less than 0.2 seconds for time travel query, and takes about 0.1 minutes for temporal join query. The query performance is pretty good, compared against the strong competitors adapted from state-of-the-art methods. In brief, for time travel and temporal aggregation queries, the proposed method (i.e., ITISS) is about 3× faster than NISS, and 4× faster than OcRT, respectively. For temporal join query, the proposed method is about 5× faster than NISS, and 7× faster than OcRT, respectively.

On the synthetic dataset (which is larger than the real dataset mentioned before), the performance gap between our proposed method and the competitors are much more obvious. In brief, as for time travel and temporal aggregation queries, our solution is 3$\sim$7 times faster than OcRT. And it outperforms NISS about one order of magnitude on both *runtime and throughput* when dataset size $|D|$ ranges from 10^6 to 4×10^9 records; especially, it outperforms NISS near to two orders of magnitude when $|D| = 4 \times 10^9$. For the temporal join query, our method is more than ten times faster than OcRT when the dataset size is 10×10^6. On the other hand, when the size of dataset is below 6×10^6 records, ITISS is about 2 orders of magnitude faster than NISS. Also, we can see that the performance of our framework drops much slower than that of competitors.

In summary, our method has better scalability and query performance, compared against the competitors. These results consistently demonstrate the superiority of our proposed method.

7 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. Based on the proposed framework, we developed targeted algorithm for time travel, temporal aggregation and temporal join queries, respectively. Besides, we 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 the superiorities of our solution.

Acknowledgements This work was supported by the NSFC (61872235, 61729202, 61832017, U1636210, U61811264, 61832013 and 61672351), and the National Key Research and Development Program of China (2018YFC1504504, 2016YFB0700502 and 2018YFB1004400).

References

1. Postgres 9.2 highlight - range types. http://paquier.xyz/postgresql-2/postgres-9-2-highlight-range-types, 2017.
2. Temporal tables. https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables, 2017.
3. Workspace manager valid time support. https://docs.oracle.com/cd/B2835901/appdev.111/b28396, 2017.
4. I. Ahn and R. T. Snodgrass. Performance evaluation of a temporal database management system. In *SIGMOD*, pages 96–107, 1986.
5. L. Alarabi and M. F. Mokbel. A demonstration of st-hadoop: A mapreduce framework for big spatio-temporal data. *PVLDB*, 10(12):1961–1964, 2017.
6. L. Alarabi, M. F. Mokbel, and M. Musleh. St-hadoop: A mapreduce framework for spatio-temporal data. In *SSTD*, pages 84–104. Springer, 2017.
7. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.
8. C. Bettini, X. S. Wang, E. Bertino, and S. Jajodia. Semantic assumptions and query evaluation in temporal databases. In *SIGMOD*, pages 257–268, 1995.
9. R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. R-tree based indexing of now-relative bitemporal data. In *VLDB*, pages 345–356, 1998.

10. M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.
11. X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. Spatial keyword querying. In *ER*, pages 16–29, 2012.
12. B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. In *ICDE*, pages 90–101, 2012.
13. L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.
14. L. Chen, S. Shang, B. Yao, and K. Zheng. Spatio-temporal top-k term search over sliding window. *World Wide Web*, pages 1–18, 2018.
15. K. Cheng. On computing temporal aggregates over null time intervals. In *DEXA*, pages 67–79, 2017.
16. R. Elmasri, G. T. J. Wuu, and Y. Kim. The time index: An access structure for temporal data. In *VLDB*, pages 1–12, 1990.
17. 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.*, 35(1):28–33, 2012.
18. D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.
19. J. A. G. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass. Parallel algorithms for computing temporal aggregates. In *ICDE*, pages 418–427, 1999.
20. S. Gollapudi and D. Sivakumar. Framework and algorithms for trend analysis in massive temporal data sets. In *CIKM*, pages 168–177, 2004.
21. H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *ICDE*, pages 336–344, 1991.
22. S. Günnemann, H. Kremer, C. Laufkötter, and T. Seidl. Tracing evolving subspace clusters in temporal climate data. *Data Min. Knowl. Discov.*, 24(2):387–410, 2012.
23. M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. Outlier detection for temporal data: A survey. *IEEE Trans. Knowl. Data Eng.*, 26(9):2250–2267, 2014.
24. C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Trans. Knowl. Data Eng.*, 11(1):36–44, 1999.
25. 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*, pages 471–482, 2015.
26. 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*, pages 1173–1184, 2013.
27. N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231, 1995.
28. G. Kollios and V. J. Tsotras. Hashing methods for temporal data. *IEEE Trans. Knowl. Data Eng.*, 14(4):902–919, 2002.
29. H. G. Lakshminarasimhan. Processing spatio-temporal data on map-reduce. pages 57–59. Springer, 2014.
30. W. Le, F. Li, Y. Tao, and R. Christensen. Optimal splitters for temporal and multi-version databases. In *SIGMOD*, pages 109–120, 2013.
31. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.
32. T. Y. C. Leung and R. R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *VLDB*, pages 383–394, 1992.
33. F. Li, K. Yi, and W. Le. Top-k queries on temporal data. *VLDB J.*, 19(5):715–733, 2010.
34. M. Li, L. Chen, G. Cong, Y. Gu, and G. Yu. Efficient processing of location-aware group preference queries. In *CIKM*, pages 559–568, 2016.
35. C. Loglisci, M. Ceci, and D. Malerba. A temporal data mining framework for analyzing longitudinal data. In *DEXA*, pages 97–106, 2011.
36. D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *ICDE*, page 35, 2006.
37. H. Lu, B. C. Ooi, and K. Tan. On spatially partitioned temporal join. In *VLDB*, pages 546–557, 1994.
38. H. Lu, B. Yang, and C. S. Jensen. Spatio-temporal joins on symbolic indoor tracking data. In *ICDE*, pages 816–827, 2011.
39. P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *VLDB J*, 8(3-4):199–221, 2000.

40. G. Özsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.
41. S. Ramaswamy. Efficient indexing for constraint and temporal databases. In *ICDT*, pages 419–431, 1997.
42. J. F. Roddick and M. Spiliopoulou. A survey of temporal knowledge discovery paradigms and methods. *IEEE Trans. Knowl. Data Eng.*, 14(4):750–767, 2002.
43. C. M. Saracco. A matter of time: Temporal data management in db2 10. Technical report, IBM, 2012.
44. A. Segev and H. Gunadhi. Event-join optimization in temporal relational databases. In *VLDB*, pages 205–215, 1989.
45. S. Shang, L. Chen, C. S. Jensen, J.-R. Wen, and P. Kalnis. Searching trajectories by regions of interest. *IEEE Trans. Knowl. Data Eng.*, 29(7):1549–1562, 2017.
46. S. Shang, L. Chen, Z. Wei, C. S. Jensen, J.-R. Wen, and P. Kalnis. Collective travel planning in spatial networks. *IEEE Trans. Knowl. Data Eng.*, 28(5):1132–1146, 2016.
47. S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.
48. S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Parallel trajectory similarity joins in spatial networks. *VLDB J.*, 27(3):395–420, 2018.
49. S. Shang, L. Chen, K. Zheng, S. J. Christian, Z. Wei, and P. Kalnis. Parallel trajectory to location join. *IEEE Trans. Knowl. Data Eng.*, pages 1–14, online first, 2019.
50. S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, pages 156–167, 2012.
51. S. Shang, R. Ding, K. Zheng, C. S. Jensen, P. Kalnis, and X. Zhou. Personalized trajectory matching in spatial networks. *VLDB J.*, 23(3):449–468, 2014.
52. S. Shang, J. Liu, K. Zheng, H. Lu, T. B. Pedersen, and J.-R. Wen. Planning unobstructed paths in traffic-aware spatial networks. *GeoInformatica*, 19(4):723–746, 2015.
53. S. Shang, K. Zheng, C. S. Jensen, B. Yang, P. Kalnis, G. Li, and J.-R. Wen. Discovery of path nearby clusters in spatial networks. *IEEE Trans. Knowl. Data Eng.*, 27(6):1505–1518, 2015.
54. D. Son and R. Elmasri. Efficient temporal join processing using time index. In *SSDBM*, pages 252–261, 1996.
55. 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.*, 31(1):32–64, 2017.
56. X. S. Wang, S. Jajodia, and V. S. Subrahmanian. Temporal modules: An approach toward federated temporal databases. In *SIGMOD*, pages 227–236, 1993.
57. R. T. Whitman, M. B. Park, B. G. Marsh, and E. G. Hoel. Spatio-temporal join on apache spark. In *SIGSPATIAL*, pages 1–10. ACM, 2017.
58. D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.
59. Y. Xu, L. Chen, B. Yao, S. Shang, S. Zhu, K. Zheng, and F. Li. Location-based top-k term querying over sliding window. In *WISE*, pages 299–314. Springer, 2017.
60. J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *ICDE*, pages 51–60, 2001.
61. Y. Yang and K. Chen. Temporal data clustering via weighted clustering ensemble with different representations. *IEEE Trans. Knowl. Data Eng.*, 23(2):307–320, 2011.
62. B. Yao, W. Zhang, Z. Wang, Z. Chen, S. Shang, K. Zheng, and M. Guo. Distributed in-memory analytics for big temporal data. In *DASFAA*, pages 549–565, 2018.
63. Y. Yuan, G. Wang, L. Chen, and H. Wang. Efficient keyword search on uncertain graph data. *IEEE Trans. Knowl. Data Eng.*, 25(12):2767–2779, 2013.
64. Y. Yuan, G. Wang, L. Chen, and H. Wang. Graph similarity search on large uncertain graph databases. *VLDB J.*, 24(2):271–296, 2015.
65. Y. Yuan, G. Wang, J. Y. Xu, and L. Chen. Efficient distributed subgraph similarity matching. *VLDB J.*, 24(3):369–394, 2015.
66. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX*, pages 15–28, 2012.
67. D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM Trans. Database Syst.*, 33(2):12:1–12:39, 2008.

68. D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *ICDE*, pages 103–113, 2002.
69. S. Zhang, Y. Yang, W. Fan, L. Lan, and M. Yuan. Oceanrt: real-time analytics over large temporal data. In *SIGMOD*, pages 1099–1102, 2014.
70. K. Zhao, L. Chen, and G. Cong. Topic exploration in spatio-temporal document collections. In *SIGMOD*, pages 985–998, 2016.
71. K. Zhao, Y. Liu, Q. Yuan, L. Chen, Z. Chen, and G. Cong. Towards personalized maps: Mining user preferences from geo-textual data. *PVLDB*, 9(13):1545–1548, 2016.