Scalable temporal clique enumeration

Kaijie Zhu[1,2], George Fletcher[1], Nikolay Yakovets[1], Odysseas Papapetrou[1]
[1] TU Eindhoven, Netherlands
[2] NDSC, Zhengzhou, China
{k.zhu,g.h.l.fletcher,hush,o.papapetrou}@tue.nl

Yuqing Wu
Pomona College
Claremont, California, USA
Melanie.Wu@pomona.edu

ABSTRACT

We study the problem of enumeration of all k-sized subsets of temporal events that mutually overlap at some point in a query time window. This problem arises in many application domains, e.g., in social networks, life sciences, smart cities, telecommunications, and others. We propose a start time index (STI) approach that overcomes the efficiency bottlenecks of current methods which are based on 2-way join algorithms to enumerate temporal k-cliques. Additionally, we investigate how precomputed checkpoints can be used to further improve the efficiency of STI. Our experimental results demonstrate that STI outperforms the state of the art by a wide margin and that our checkpointing strategies are effective.

CCS CONCEPTS

• **Information systems** → **Data structures**; **Join algorithms**; **Spatial-temporal systems**;

KEYWORDS

temporal join, temporal clique, query processing, checkpoints

ACM Reference Format:
Kaijie Zhu[1,2], George Fletcher[1], Nikolay Yakovets[1], Odysseas Papapetrou[1] and Yuqing Wu. 2019. Scalable temporal clique enumeration. In *16th International Symposium on Spatial and Temporal Databases (SSTD'19), August 19–21, 2019, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340964.3340987

1 INTRODUCTION

The problem. We study the *temporal k-clique enumeration* problem: given (1) a collection of data objects where each object has an associated time window; (2) a query time window; and, (3) a non-negative integer k, enumerate all k-sized subsets of objects which are concurrent in the query time window (i.e., all mutually overlapping at some time point in the query window). This problem arises in a wide range of applications. Some illustrative examples follow.

- In case of disease eruption and its transmission in a community, find all groups of k people whose infectious periods all pairwise overlap in a given timeframe.
- For a deeper understanding of scientific collaborations in a bibliographic database, find all groups of k people who have tightly collaborated with each other at the same time, in a given time period.
- For calling a meeting, given the availability of one or more timeslots per committee member, determine possible meeting schedules in a given time period, based on the need to reach a quorum of k available members.
- A typical lion pride consists of 8 to 9 adults whereas a large pride consists of 30 to 40 adults.[1] Most social animal groups likewise have well-defined bounds on membership size. In an ecological animal database, identify large lion prides visiting a particular location in a given time window (i.e., $k = 30$ adult lions which temporally co-occur at the location).
- For analytics on a temporal graph (i.e., a graph where each edge in the graph has an associated time window), identify k-sized subgraphs which temporally co-occur in a given time window, where k is the target subgraph size. For example, towards targeted recommendations in a social network, identify small groups of people ($k < 5$) who are all mutually socially connected in a given time window.

Viewing time windows as intervals, the k-clique problem also arises as a basic challenge in the context of spatial and uncertain data management, e.g., [9].

To our knowledge, the general temporal clique problem has not been identified and studied before. The special case when $k = 2$ has been well-studied in the literature as the *interval join* problem [5, 11, 18]. Although current competitive methods for interval joins can be easily adapted to clique enumeration, they still suffer from scalability issues such as unnecessary scanning of records (highlighted in Section 3 below) when applied to large data sets occurring in practice.

Our contributions. In our work, we directly address the scalability issues found in the state of the art. In particular:

- Based upon a careful analysis of the weaknesses of state of the art methods, we propose a novel method, the start time index (STI) algorithm, to process clique enumeration efficiently (Sections 2-4).
- We develop checkpoint mechanisms to further improve query processing in STI (Section 5). We discuss four checkpointing strategies and highlight their benefits. In addition to STI, these strategies are of independent interest and could also be applied in combination with other state of the art methods.

[1] https://cbs.umn.edu/research/labs/lionresearch/social-behavior

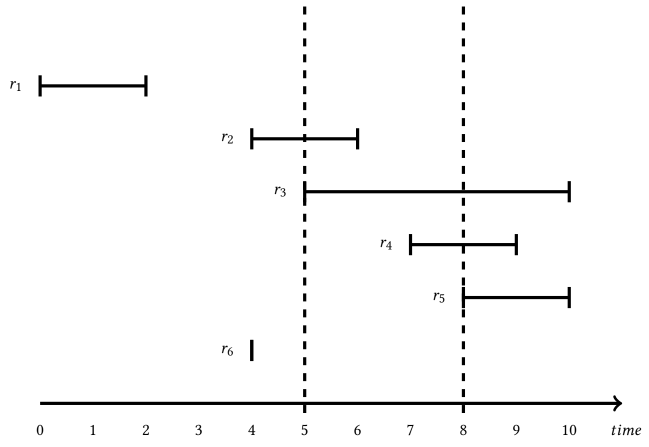

Figure 1: An example of time intervals. Temporal relation R_{ex} and query $[5, 8]$.

- We present the results of an in-depth experimental study which demonstrates the significant improvements in scalability and performance introduced by our new methods (Section 6).

Our methods give rise to several interesting avenues for further research in temporal data analytics. We conclude the paper with a summary of our findings in Section 7.

2 PRELIMINARIES

In this section, we introduce basic concepts and definitions.

Temporal data. A *time window* is an ordered pair of non-negative integers $[i, j]$ such that $i \leq j$. We refer to i and j as *time-stamps.* We say time window $[i, j]$ *contains* time window $[k, l]$ if $k \geq i$ and $l \leq j$, which we denote by $[k, l] \sqsubseteq [i, j]$. We say $[i, j]$ and $[k, l]$ *overlap* if $i \leq l$ and $k \leq j$, i.e., there is a time window w contained in both $[i, j]$ and $[k, l]$. The *length* of window $w = [i, j]$ is the value $|w| = j - i$

We consider finite sets R equipped with an injective function $rid : R \longrightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. We assume that for each element $r \in R$ an associated time window $[\overline{r}, \underline{r}]$ is given. We will refer to such collections R as *temporal relations* or just *relations.* We will refer to the elements of temporal relations as *records*, and refer to the time windows associated with records as *intervals.* We say record r is *active* at timestamp t if $[t, t] \sqsubseteq [\overline{r}, \underline{r}]$.

The domain of relation R is the shortest length time window $D = [\overline{R}, \underline{R}]$ such that, for every $r \in R$, it holds that $[\overline{r}, \underline{r}] \sqsubseteq [\overline{R}, \underline{R}]$.

Temporal cliques. Let S be a relation and w be a time window. If there exists a time window $t \sqsubseteq w$ such that for every $s \in S$ it is the case that $t \sqsubseteq [\overline{s}, \underline{s}]$, then we say the elements of S form a *temporal* $|S|$*-clique in* w, where $|S|$ is the number of elements of S.

As a simple running example, consider temporal relation $R_{ex} = \{r_1 : [0, 2], r_2 : [4, 6], r_3 : [5, 10], r_4 : [7, 9], r_5 : [8, 10], r_6 : [4, 4]\}$ and time window $w = [5, 8]$, visualized in Figure 1. There is exaclty one temporal 3-clique in w, namely, $\{r_3, r_4, r_5\}$.

Problem statement. We study the problem of "temporal k-clique enumeration", defined as follows. Let R be a temporal relation, k be a positive integer, and $q = [qstart, qstop]$ be a time window (i.e., a query). Enumerate all $S \subseteq R$ where S is a temporal k-clique in q.

Concurrent sets and Living history windows. Given a temporal relation R and a timestamp t, we call the largest temporal clique $S \subseteq R$ in $[t, t]$ the *Concurrent Set at* t, denoted $CS(t)$. In other words, $CS(t)$ consists of all records that are active at timestamp t. The size of $CS(t)$ is denoted by $CSS(t)$.

We study methods for clique enumeration based on constructing $CS(qstart)$ and then maintaining the set as we scan forward in time to *qstop*. We call the time window scanned during constructing $CS(qstart)$ the *Concurrent Set Construction Window.* Note that this window is algorithm-dependent. For example, in the most naive method, the construction window is $[\overline{R}, qstart]$, which could be unnecessarily costly in practice.

Given a temporal relation R and a timestamp t, the *earliest concurrent* of t is the timestamp $eC(t)$ corresponding to the earliest start time of those records that are still active at t, i.e.,

$$eC(t) = \begin{cases} \text{undefined} & \text{if } CS(t) = \emptyset \\ \min_{r \in CS(t)} \overline{r} & \text{otherwise} \end{cases}$$

Note that $eC(t)$ only makes sense when $CS(t) \neq \emptyset$. We call the window $[eC(t), t]$ the *Living History Window* of t, denoted $LHW(t)$.

Continuing our running example (Figure 1), we have $eC(8) = 5$ and $LHW(8) = [5, 8]$, because r_3 is still active at $t = 8$.

Methods for constructing $CS(qstart)$ at a minimum must process those records with start times which occur in $LHW(qstart)$. We consider query processing in two adjacent windows as shown in Figure 2: (1) a living history window and (2) a query window. The former one is used to construct $CS(qstart)$ while the latter one is used to enumerate cliques. With the introduction of living history, the concurrent set construction window is significantly shortened compared to the naive method.

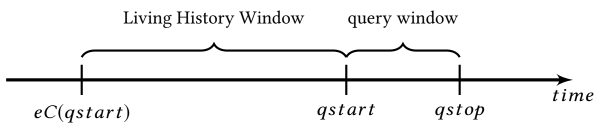

Figure 2: Query Processing via linear scan

3 RELATED WORK

Temporal joins. Research on interval join processing can be classified [12] in index-based, partition-based, and plane-sweep methods. Index-based methods construct and maintain specialized data structures in order to speed up query processing. A bi-temporal index that could be used to compute interval joins on two temporal dimensions (i.e. both system and application time) is proposed in [14]. An algorithm based on a two-layer flat index (Overlap Interval Inverted, O2i) is presented in [16]. Indexed segment tree forest (ISTF), in which the temporal nesting relationships are represented by a binary tree and joins are enumerated by searching related trees is proposed in [17]. Partition-based methods cluster intervals into smaller buckets based on their similarity, and join-processing is done for certain pairs of buckets to reduce the unproductive evaluations. Dignos et al. [10] proposed a self-adjusting algorithm named OIP divides intervals into k equal-sized consecutive granules and a method to a best k is also proposed, which could lead to a minimal compromise of query cost and unproductive join ratio when timeline is divided into the same number of granules. Cafagna et al. [7] proposed DIP to divide temporal relation into partitions containing non-overlapped tuples, which also reduces number of the unproductive join operations in evaluation.

Currently, the best performing solutions for interval joins are based on plane-sweep methods [5]. Piatov et al. [18] proposed two memory plane sweep-based interval join algorithms EBI and LEBI based on endpoint index, which outperform OIP and prior plan-sweep methods. Bouros et al. [5] proposed two optimized algorithms based on forward scan named gFS and bgFS. We next discuss in detail these two general approaches, which are the state of the art methods.

Endpoint-based Index EBI [18] is an internal-memory-based plane-sweep algorithm for processing an interval join between relations R and S. In EBI, each record r with associated time window $[\overline{r}, \underline{r}]$ is represented as a pair of endpoint events, where each event represented by tuple $(ti, ty, rid(r))$. Here, ti is the timestamp of an endpoint and is either $\overline{r}$ or $\underline{r}$, ty is the endpoint type and should be either $start$ or $stop$, and $rid(r)$ is the index of the record r. Given a pair of temporal relations R and S, their endpoint index EI^R and EI^S is then constructed. The events are sorted by their ti in ascending order. As for join-processing, EI^R and EI^S are scanned concurrently from the beginning and each event is accessed forwardly. During the scan for each index, a dedicated structure named active list is maintained to store the concurrent set of relation in real time, denoted as A^R and A^S. The active list is updated depending on the type of scanned endpoint. And for each scanned $start$ endpoint, EBI matches it with all the records in opposite active list to produce joins.

EBI can also be used for k-clique enumeration through a straightforward modification. We extend each event tuple into the following format: $(ti, ty, rid(r), eC(ti))$. The new fourth position records the earliest concurrent of ti. It could be updated in the end of index initialization through a full scan. Such an extension aims at a fast location on the living history window of queries. Based on this, a query $[qstart, qstop]$ could be processed through a linear scan: firstly, EBI determined LHW(qstart) according to extended tuples in the index. Next, starting from the beginning of the LHW(qstart), an active list *Active* is maintained in the same way as in original EBI. Enumeration is not carried out until the scanning cursor moves into the query window. In other words, the time an tuple with higher ti than $qstart$ is encountered. To begin with, EBI firstly enumerates all the k-sized subsets of *Active* to output all cliques in which all intervals start before $qstart$. Then every time a start event $(\overline{r}, start, rid(r))$ is scanned, a batch of joins could be produced by matching the event with all $(k-1)$-sized subsets of *Active*. The linear scan is stopped when the cursor moves out of the query window. In this way, all k-cliques are produced.

Forward Scan Algorithm Compared to EBI, forward scan (FS) [6] directly performs a linear scan on relations without using dedicated structures (i.e., active list). Relations R and S in forward scan algorithm are sorted by the start time of records. Two linear scans are carried out on from the beginning of relation and stop each time at a new record. For each scanned record in a relation, FS matches it with all overlapping records in the opposite relation. In this way, all pairs of interval joins are produced. gFS and bgFS [5] are improved versions of FS. In gFS, similar consecutive intervals are grouped and matched with overlapping intervals in opposite relation instead of comparing pairs of intervals one by one. In bgFS, the domain of relations are segmented into equal-sized dedicated buckets and intervals are put in corresponding buckets based on their start time. With the bucket index, the comparisons for join-matching need merely be made between interval groups in one relation and buckets in another. The two extensions reduce the cost of comparison and scanning in original FS.

gFS and bgFS can also be adapted directly to enumerate k-cliques. Firstly, each record should be extended with earliest concurrent and maintained just like in EBI. Secondly, the condition of grouping should be modified as follows. Given the relation R and for any two time instances t_1 and t_2 with $t_1 < t_2$, if no interval from R ends in $[t_1, t_2]$, then all intervals from R that start in $[t_1, t_2]$ can be grouped together to reduce comparisons. In other words, we can group all intervals which have succeeding endpoints of the same type (either start or end).

Inefficiencies of current plane-sweep methods. EBI is inefficient in several aspects. For *index-scanning*, though the living history window is used, the scanning range is still much larger than the given query window. In the worst case, the scan could start from the leftmost event tuple, just like the time when the notation of living history window is not proposed. Besides, decoupling endpoints could lead the scanning cost to be much larger than the number of events that actually involve enumeration: (1) For records starting and also stopping in the scanning range, the scanning cost doubles. (2) For records starting before scanning range but stopping in, their stop event tuples are still to be scanned though they do not involve the enumeration.

For *active-list maintenance*, either an insertion or a deletion will be performed for each encountered tuple during the scan. Some of these performed before $qstart$ are irrelevant to the results. In the worst case, the number of operations performed on irrelevant tuples can be significantly larger than that of events which contribute to the final query results.

Finally, for *index storage*, as number of event tuples doubles the number of records in relation, the storage cost increases significantly.

Next we turn to gFS and bgFS, which introduce inefficiencies by the production of duplicate index-scanning. Without decoupling endpoints in gFS and bgFS, the living history window scanning cost does not double. However, the worst case of living history window still exists. Besides, many records tend to be scanned more than twice because of the inconsistent references in grouping and enumeration. In other words, the scanning range in grouping depends on the smallest stop time of grouped records while the scanning range in enumeration depends on the largest stop time. Therefore, in FS family of algorithms, the existence of long intervals can potentially introduce significant redundant scanning cost.

Temporal checkpoints. In [14, 15] checkpoints are implemented as bit vectors to represent the status of relation at certain times. However, currently there is no research on checkpointing strategies. To the best of our knowledge, we are the first to carry out the research on checkpoints for clique enumeration.

4 OUR METHOD: START TIME INDEXING

In this work, we propose Start Time Index (STI) with the aim to provide more efficient temporal k-clique enumeration. Instead of splitting a single interval into two event tuples, only the start-event tuple in a form of [$\overline{r}$, $\underline{r}$, $rid(r)$, $eC(\overline{r})$] is used to represent each record in an STI. Given a temporal relation R, an STI index SI is constructed as follows. First, start-event tuples are inserted into a B+tree sorted by their start time. Each tuple is initialized as [$\overline{r}$, $\underline{r}$, $rid(r)$, -1]. Then, after all intervals have been inserted, SI is scanned and the last position in each start-event tuple is updated to $eC(\overline{r})$.

We provide the following API for performing a look-up in an STI:

- $getEntry(t)$ ($getRecent(t)$): given a timestamp t, retrieve the first start-event tuple r with the smallest (largest) timestamp among all index entries that satisfy $t \leq \overline{r}$ ($t > \overline{r}$). Return the first start-event tuple of an STI if no such tuple exists.
- $startScan(r)$: start a linear scan of start event tuples along the index starting from r and return a cursor sc for fetching each tuple.
- $nextEntry(sc)$: retrieve the next start-event tuple of scan sc.
- $stopScan(sc)$: stop the scan sc in an STI.

By using the B+tree structure, the complexity of $getEntry(t)$ and $getRecent(t)$ is $O(\log N)$ where N is the number of entries in a B+tree. The complexity of the other methods is $O(1)$.

Query processing. Using STI, the queries are processed as follows. For each query in a workload, the algorithm involves two B+tree look-ups and one linear scan. The look-ups aim to identify the living history window for a linear scan. The first look-up calls $getRecent(qstart)$ to retrieve the most recent tuple r before $qstart$ ($\overline{r}$ is marked as t_0 in Figure 3). The second look-up uses $eC(t_0)$ as the search key in order to locate the leftmost start-event tuple in living history window. Note that we identified [$eC(t_0)$, $qstart$] as living history window *approximately* rather than [$eC(qstart)$, $qstart$] as there might be no events starting at $qstart$. We call the [$eC(t_0)$, $qstart$] an *approximate living history window.*

Just like in EBI, during the linear scan, STI approach maintains the in-memory active list ($Active$) to record the concurrent set in real time. The tuples in $Active$ are sorted by their stop time in the ascending order. We define the following operations to maintain $Active$:

- $insActive(Active, r)$: insert the start-event tuple r into $Active$;
- $delActive(Active, t)$: delete all start-event tuples r s.t. $t > \underline{r}$ from $Active$;
- $enumActive(Active, k)$: generate all temporal k-clique subsets over the elements of $Active$;
- $incEnumActive(Active, r, k)$: first insert r into $Active$, then generate all temporal k-cliques over the elements in $Active$ which contains an occurrence of r.

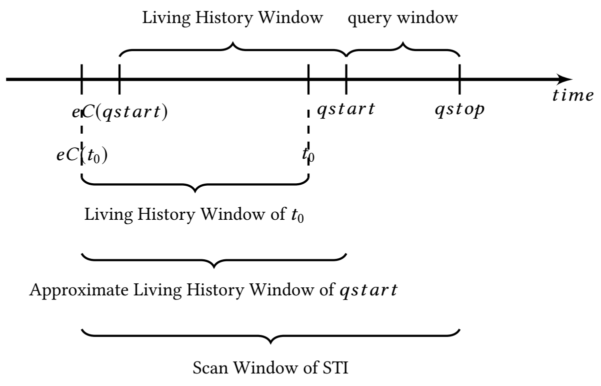

Figure 3: Query Processing using STI, where t_0 is the start time of $getRecent(qstart)$

The procedure of the STI algorithm is shown in Algorithm 1. A linear scan starts from the leftmost tuple in approximate living history window of $qstart$ and scans forward. $Active$ is real-time maintained by inserting newly scanned start-event tuple $curr$ and deleting the expired tuples. When the first start-event tuple in a query window is scanned, all k-clique subsets in $Active$ are returned. From then on, every scanned start-event tuple $curr$ would be matched with all (k-1)-cliques in $Active$ until either (1) the newly scanned $curr$ starts after the query window, or (2) the end of the relation is reached. These two situations show that all involved start-event tuples have been scanned and the algorithm should be halted. If there are no start-event tuples in a query window, the algorithm directly enumerates all the k-clique subsets in $Active$ as the result. This way, all k-cliques in [$qstart$, $qstop$] of relation R are enumerated.

Consider again our running example relation from Section 2 (Figure 1). The STI constructed for R_{ex} is the list

$$SI = \{(0, 2, r_1, 0), (4, 4, r_6, 4), (4, 6, r_2, 4), (5, 10, r_3, 4), (7, 9, r_4, 5), (8, 10, r_5, 5)\}.$$

To enumerate the 2-cliques in $q = [5, 8]$, the algorithm firstly determines [4, 5] to be the approximate living history window, by retrieving $eC(5) = 4$ from $(5, 10, r_3, 4)$. Starting from the beginning of the approximate living history window, the processing procedure in shown in Table 1.

Complexity. Contrary to FS, duplicate scanning would not happen in STI. Compared to EBI with decoupling endpoints, STI requires space for storing entries and internal nodes in the B+Tree. The scanning cost in STI is improved: (1) for records starting and also stopping in the scanning range, the scanning cost is halved because one record is represented by a single tuple rather than a pair; (2) for records starting before the scanning range but not ending, no additional scanning cost is introduced because the start-event tuples are sorted by start time, which makes it impossible for such records to appear in the scanning range. Additionally, operations on irrelevant tuples in active-list maintenance are completely avoided since their stop time are compared with $qstart$. Finally, extra scanning cost introduced by the approximation of the living history window is minor compared to the scanning benefits gained from the STI,

Algorithm 1: STI temporal k-clique enumeration

```
   Input: relation R (with STI index $SI$), $qstart$, $qstop$, k
   Output: outstream of temporal k-cliques in $[qstart, qstop]$ of R
1  $startTS \leftarrow eC(\overline{I.getRecent(qstart)})$
2  $Active \leftarrow \emptyset$
3  $curr \leftarrow SI.getEntry(startTS)$
4  $inRange \leftarrow false$
5  $sc \leftarrow SI.startScan(curr)$
6  $result \leftarrow \emptyset$
7  while $curr \neq NULL$ do
8      if $\overline{curr} < qstart$ then
9          if $\underline{curr} >= qstart$ then
10             $insActive(Active, curr)$
11     else if $\overline{curr} \leq qstop$ then
12         if $inRange = false$ then
13             $delActive(Active, qstart)$
14             $result \leftarrow result \cup enumActive(Active, k)$
15             $inRange \leftarrow true$
16         $delActive(Active, \overline{curr})$
17         $result \leftarrow result \cup incEnumActive(Active, curr, k)$
18     else
19         if $inRange = false$ then
20             $delActive(Active, qstart)$
21             $result \leftarrow result \cup enumActive(Active, k)$
22         break;
23     $curr \leftarrow getNext(sc)$
24 if $inRange = false$ then
25     $delActive(Active, qstart)$
26     $result \leftarrow result \cup enumActive(Active, k)$
27 $SI.stopScan(sc)$
```

Table 1: STI Example

Tuple	*Active*	*Operation*
(4, 4, r_6, 4)	$\{r_6\}$	Insert r_6
(4, 6, r_2, 4)	$\{r_2, r_6\}$	Insert r_2
(5, 10, r_3, 4)	$\{r_2, r_3\}$	Delete r_6; Insert r_3; Enumerate (r_2, r_3);
(7, 9, r_4, 5)	$\{r_3, r_4\}$	Delete r_2; Insert r_4; Enumerate (r_3, r_4);
(8, 10, r_5, 5)	$\{r_3, r_4, r_5\}$	Insert r_5; Enumerate (r_3, r_5), (r_4, r_5);

especially in datasets with extremely long intervals. However, we should note that the case of the worst living history still exists. That is, the living history window could still be very large in its absolute temporal measurement.

Maintenance. Maintenance of STI under insertions and deletions of records in R incurs the costs of B+tree maintenance and the cost of updating earliest concurrent values of affected entries. When a new tuple r' is inserted, $eC(\overline{r'})$ could be obtained by using the information provided by its adjacent tuples. In addition, we traverse each tuple $r \in [\overline{r'}, \underline{r'}]$ and update $eC(\overline{r})$ to $\overline{r'}$ if $eC(\overline{r}) > \overline{r'}$. When an existing tuple r' is deleted, we carry out a linear scan on tuples in $[\overline{r'}, \underline{r'}]$ with *Active* maintained and update $eC(\overline{r})$ to $\min_{r \in Active} \overline{r}$ if $eC(\overline{r}) = \overline{r'}$.

Summarizing, STI addresses all the inefficiencies of EBI and FS except the long living history window problem. In next section, we present our methods on how to solve this problem.

5 OPTIMIZED STI

In this section, we introduce Start Time Index with Checkpoints (STI-CP), which is a variant of STI enhanced with *checkpoints*, aiming to speed up the processing of long living history windows. A single checkpoint is a dedicated structure composed of a timestamp c and $CS(c)$, which represents the concurrent set at timestamp c.[2] Given a temporal relation R, necessary index structures in STI-CP include: (1) a start time index SI and (2) a set of checkpoints $C = \{c_1, c_2, \ldots, c_k\}$. Then, the linear scan in STI could start from the timestamp of the latest checkpoint which timestamp is smaller than $qstart$ rather than from the beginning of living history window. In the best case, $CS(qstart)$ could be directly obtained. The major difference between STI-CP and STI are:

- In STI-CP, an additional data structure that stores the concurrent set for each checkpoint in C is maintained. After the STI is constructed, a checkpointing procedure proceeds to select some checkpoints in temporal domain and store them in dedicated structure for further use.
- In STI-CP, $LHW(t)$ starts from $max(eC(t), lateCP(t))$ where $lateCP(t) = \max_{c \in C \wedge c \leq t}(c)$. We call such timestamp a *history pointer* of time t, denoted $HP(t)$. That is, $LHW(t)$ in STI-CP is $[HP(t), t]$ rather than $[eC(t), t]$ in STI.

The STI-CP algorithm is shown in Algorithm 2. The following functions are provided in STI-CP in addition to those in STI:

- *getHistoryPt*(t): Given a timestamp t, returns the history pointer of t. It may be either $eC(t)$ or a timestamp of a checkpoint.
- *isCP*(t): Given a timestamp t, returns true if $t \in C$ and false otherwise.
- *getConcurSet*(t): Given a time-stamp t, returns $CS(t)$ if $t \in C$ and $\emptyset$ otherwise.

Algorithm 2: STI-CP temporal clique enumeraion

```
   Input: relation R (with STI-CP index $I$), $qstart$, $qstop$, k
   Output: outstream of temporal k-cliques in $[qstart, qstop]$ of R
1  $startTS \leftarrow getHistoryPt(\overline{I.getRecent(qstart)})$
2  $Active \leftarrow getConcurSet(startTS)$
3  ... continue on with Algorithm 1 starting from line 3.
```

Continuing our running example from Section 4, if a checkpoint c is set at $t = 5$, the scanning of living history window [4, 5] becomes unnecessary, leading to a saving in scanning costs.

Maintenance. Checkpoints could also be used to accelerate the maintenance of STI, as discussed in Section 4. As for the maintenance of checkpoints themselves, an insertion of tuple r' would lead to inserting r' into $CS(c)$ for each $c \in [\overline{r'}, \underline{r'}]$. Similarly, a deletion of tuple r' would lead to deleting it from the same group of checkpoints.

The effectiveness of STI-CP depends on the selected checkpoints (CPs). To improve the efficiency in STI-CP, we concentrate on the problem of checkpointing, which could be formalized as follows:

Checkpointing Problem. Given a set Q of m queries in a workload, a set T of n checkpoint candidates, and storage budget B,

[2]For brevity, c denotes a checkpoint at timestamp c

a checkpointing problem $G =< Q, T, B >$ aims to obtain a solution $C = \{c_1, c_2, \dots, c_k\}$ such that $\sum_{i=1}^{k} CSS(c_i) \leq B$, $k \leq n$, and $\forall c_i \in T$.

One can easily show NP-completeness of the corresponding optimal checkpointing problem by producing a reduction to a 0-1 knapsack problem. In the rest of this section, we propose several heuristics for checkpointing, which could obtain the effective solutions at low cost.

The most basic heuristic method is based on *random* checkpointing. This method is easy to implement, but it does not make a good use of the budget as random placement of CPs might do little to improve query times. With the aim of obtaining more effective checkpointing strategies, we consider data distribution and query workload and propose four checkpointing strategies classified in two broad categories: *data-aware* and *workload-aware*.

Note that the initialization of the CP index consists of two phases: the CP selection and insertion phase. The complexity of the first phase depends on the checkpointing strategy while that of the second phase is strategy-independent.[3] So, for each strategy, we only analyze the complexity of its CP selection.

5.1 Data-aware strategies

Our first checkpointing strategy is called *binary* strategy since it selects the CPs in a binary, breadth-first manner, until the storage budget is consumed. In every round, pairs of records with the largest distance in which no CP exists is retrieved, and the middle point of the pairs in the duration is selected as the position to set a new CP. In this way, the CP distribution becomes even and pairs of CPs that are too close to each other (and, thus, potentially wasteful) could be avoided.

We define two types of distances that could either be used in a binary strategy: (1) event and (2) temporal distance. Event distance reflects the distance measured in the number of events, while temporal distance reflects the distance measured in time. By considering temporal distance, we capture the burstiness [13] of temporal data.

The complexity of both binary methods is $O(k \log k)$, where k is the number of CPs. As the event and temporal distance are respectively tuple and timestamp-based, we define following mapping methods from timestamp to tuple:

- $first(t)$ ($last(t)$): given a timestamp t, $first(t)$ ($last(t)$) returns the number of the first (last) start-event tuple r s.t. $\overline{r} = t$. If there is no tuple starting at t, $getEntry(t)$ is returned.

Binary strategies do not consider the impact of long intervals in data. That is, extremely long intervals could influence a large number of tuples in index. Specifically, given a start-event tuple r and a long interval r', $eC(\overline{r})$ should be at least as small as $\overline{r'}$ if r' overlaps r. Hence, to process such queries, algorithm needs to start the scan from $\overline{r'}$ at least when no CP is present. Meanwhile, we should also note that queries are more likely to overlap long intervals when they are uniformly distributed.

[3]Given N index entries and k CPs, it takes $O(N \log N)$ time to scan the STI index while maintaining an active list and $O(k \log k)$ time to collect CSs and then insert them into the CP index bringing overall complexity to $O(N \log N + k \log k)$.

Based on this, we propose the *long link half* strategy, which gives priority to long intervals to be assigned CPs in order to reduce their impact. The outline of the long link half could be divided into two phases. The first phase is to construct the *link map* of the STI. Link map is a dedicated structure providing guidance for later CP selection. The construction procedure consists of two steps: first, we scan the STI to collect all *influential intervals*.

Influential Interval. Given an index SI and a start-event tuple r_0, if $\exists r \in SI$ such that $eC(r) = \overline{r_0}$, we call $[\overline{r_0}, \underline{r_0}]$ an influential interval in SI.

All collected influential intervals are sorted by $first(\overline{r_0})$ in ascending order. The collection in STI covers the whole domain. Longer influential intervals tend to have more opportunities to cover the uniformly distributed queries, which also tend to produce longer living history window.

Next, the collection is *refined* and used to build up a map. Starting from the first interval, each interval is continuously combined with the following interval into a longer interval until they no longer significantly overlap (according to some threshold u). Such combination could help to avoid the potentially wasteful CPs in later selection phase. In this way, we obtain the map that could provide us an overview of the distribution of influential intervals in STI.

Given the refined map, the second phase is to select CPs under its guidance. In every round, the longest interval is taken out from the map, and the start time of its middle event is selected as a place to set CP. The CP segments the interval into two sub-intervals and this round would be carried recursively until the budget B is consumed. The selected CPs have a high tendency to segment long intervals, which turns out to weaken the long interval impact.

The complexity of the construction of the map is $O(N + L \log L)$, where N is the number of entries in index and L is the number of link tuples in map. Considering influential intervals are collected through a single scan, we could move the collecting step into the initialization of the STI index. So the additional linear scan is unnecessary and the map construction complexity could be reduced to $O(L \log L)$. The complexity of selection is $O(k \log L)$. Hence, the total complexity of the long link half strategy is $O((L + k) \log L)$.

5.2 Workload-aware strategies

We model real-world query workloads as being composed of two parts: first, a small proportion of uniformly distributed queries, which represents the outlier behavior performed by some users. Second, a large proportion of queries distributed around several *hotspots* in the domain of a dataset. For example, considering a scheduling of the free time slots in classrooms in a university campus, most queries would aggregate in January and July since it is the time for final examinations. Using such queries, administration staff could find the pairs (or larger subsets) of classrooms available to simultaneously hold examinations. Considering a biological database recording retention period of zebras in Serengeti National Park in Africa, researchers might be interested in querying the pairs of zebras staying simultaneously in one place. Most of such queries would aggregate in the first half of each year as in the later half zebras would move to Masai Mara for abundant grass and water.

This example also implies that burstiness patterns in real world could also be a factor in aggregation.

We propose a workload-aware strategy named *query-set*, which consists of two phases: in the first phase, a batch CPs is selected for clusters based on their importance. Secondly, if budget allows, another batch is selected for the uniformly distributed queries in order to improve the global efficiency.

The basic idea for the first phase is to identify the hotspots in workload, clustering queries around each hotspot, and selecting the CPs for each cluster. Many existing works could be used to detect such clusters. Here, we choose the mean-ISI method [8] to obtain the aggregated pattern. Note that there are situations where some clusters could not receive a CP since budget B is limited, so it is necessary to determine which clusters should have a priority. For this reason, we introduce a metric named *cluster importance*, denoted as CI, to assist in making this decision.

Cluster Importance. Given a query cluster Cl and the minimal time window $[\overline{Cl}, \underline{Cl}]$ covering all $qstart$ of queries in, its cluster importance CI could be calculated as follows:

$$CI = |Cl| \cdot (LH(\overline{Cl}) + last(\underline{Cl}) - first(\overline{Cl} + 1))$$

where $|Cl|$ refers to the number of queries in Cl.

The idea of CI is to approximate the living history window scanning cost for the whole cluster. This estimation is efficient when Cl is large. We put all the query clusters in a list and sort them by their importance in descending order. Recursively, we select the cluster with the highest priority from the remaining clusters and set a checkpoint at $\overline{Cl}$, until the budget is consumed.

After the batch of initial CPs for each cluster are selected, duration of clusters would be sorted by the number of records inside in descending order. Ideas in event binary and long link half strategy are recursively applied to the re-sorted list until either (1) B is consumed or (2) the number of records in every remaining duration is shorter than a pre-configured threshold x. That is, when (2) happens, it demonstrates most of queries in the cluster could benefit from CPs so we need to stop selecting CPs for clustered queries and turn to the next phase.

In the second phase, CPs are selected in the same way as in long link half strategy, aiming at the improvement on the processing of uniform queries and full use of the remaining budget.

The complexity of clustering is $O(|Q| \log |Q|)$. The further segmentation of cluster durations takes $O((m + n) \log m)$, where m is the number of clusters in workload and n is the number of CPs selected in this step. The second phase has the same complexity as long link half strategy. So the total complexity of query-set strategy is $O(|Q| \log |Q| + (m + n) \log m + (k + L) \log L)$.

6 EXPERIMENTS

In this section, we present our experimental investigation of STI and STI-CP approaches. We aim to answer the following questions. First, we would like to know if the STI approach can outperform the other methods on a number of diverse datasets and query workloads. Second, given a storage budget, we investigate to what extent could various checkpointing strategies improve the efficiency of the STI family.

6.1 Setup

Environment. Our experiments were carried out on a server with 192GB RAM and 2 Intel(R) Xeon(R) CPU X5670 with 6 cores at 2.93GHz running a Linux operating system. We implemented the in-memory versions of STI and STI-CP approaches in C++. We use an in-memory version of BerkleyDB B+tree in which we set the page size to 8KB and use a 12-byte search key to find 8-byte data values.

Competitors. We use the k-clique enumeration versions of EBI, gFS, and bgFS as the competitors to STI. The length of buckets in bgFS is set to 1000 units of time. Since the processing of gFS and bgFS requires records to be sorted by their start time, these two algorithms are using the same B+tree as STI. That is, the difference between STI and the two forward-scan baselines is in their processing of the query window.

Query Generation. We consider two methods of query generation: (1) a uniform method, which aims to simulate the workload that has queries with their start times uniformly distributed in time, and (2) a clustering method, which aims to simulate the workload that has queries with their start times clustered in one or more *hotspots* in time. This query generation model requires three parameters to be specified: (1) the number of queries N in the workload, (2) the proportion of the size of the query window in relation to the entire time domain $l \in [0, 1]$, which determines the window size of generated queries[4], and (3) the clustering proportion $r \in [0, 1]$, which represents a proportion of clustered queries in a workload. Given r, we compute the number of queries that need to be clustered. Denoting the collection hotspots as $\{t_1, t_2, ..., t_n\}$ where t_i is the timestamp of the ith hotspot, we assume that each hotspot contains a predetermined number of queries which follow a normal distribution with $\mu = t_i$ and $\sigma = 100$.

The workload in our experiment is composed of above two categories of queries. For each workload, we generate 2000 queries in total which are then split into a training set and a test set, 1000 queries each. The training set is used for workload-aware STI-CPs to learn the clustering structures.

Types of experiments. We run two types of experiments: (1) we process queries with various query window sizes (which are determined by l in $[0, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 20]\%$), to investigate the performance of algorithms in dealing with both long and short queries; and (2) we experiment with datasets of different sizes. The largest dataset used in this experiment has 400 million records[5] to investigate the scalability of algorithms with respect to the dataset size.

For each algorithm, we use three metrics to evaluate its efficiency: the average execution time[6] for each query (i.e., its processing cost), memory consumed by indexes (i.e., its storage cost), and index construction time (i.e, its preparation cost). For processing cost, we set the timeout to 10^5 seconds, after which we consider the instance to be not competitive.

[4]We consider a special case when $l = 0$ to generate a workload of *instant* timestamp queries such that query's start time is the same as its stop time.
[5]After cleaning, the size of original file for the 400-million dataset is more than 10GB
[6]The time cost of enumeration is not included.

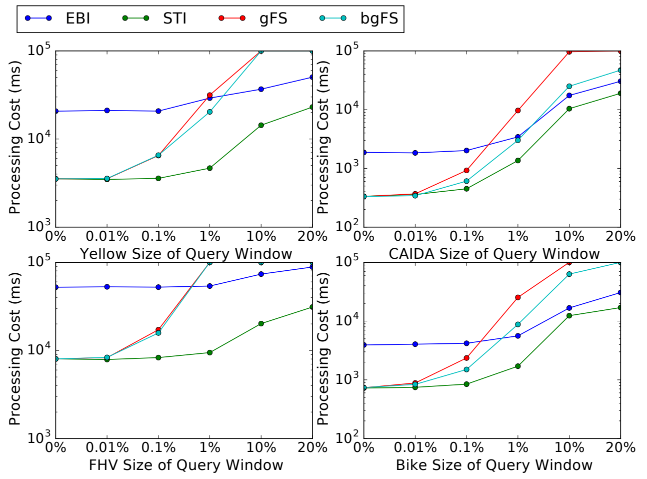

Figure 4: Performance for basic algorithms with respect to the query window size

For STI-CPs, we carry out two additional types of experiments to investigate the efficiency of various checkpointing strategies: (1) we set the budget B to [0.2, 0.4, 0.6, 0.8, 1]% of the dataset size to investigate the effect of budget size on different checkpointing strategies; (2) we set the clustering proportion r to [0.5, 0.8, 0.9, 0.95, 1] to investigate the effect of query hotspots on STI-CP evaluation.

Datasets. We consider four real-world datasets from telecommunication and transportation domains: Yellow, CAIDA, FHV, and Bike. Yellow[4] records the trips on the yellow taxi in New York City from 2010 to 2018 and each trip is labeled with an interval to represent its duration. CAIDA[1] records the anonymized passive traffic traces from Center for Applied Internet Data Analysis (CAIDA) in 2018. Each session is labeled with an interval to represent its duration. FHV[3] records the trips on free hired vehicles in New York City in the second half of 2017. Bike[2] records the trips on citi-bikes in New York City in from 2013 to 2018.

6.2 Results and Analysis

Investigation for STI. We first investigate the effectiveness of our basic algorithm, the STI algorithm. Figure 4 reports the processing cost of EBI, STI, gFS, and bgFS with respect to the size of the query window. We note that STI outperforms all of its competitors in the processing cost, especially when the size of query window increases to 0.1%, 1%, 10%, 20% of the time domain. Compared to EBI, STI has lower processing cost in both living history and query window. Compared to gFS and bgFS, STI scales better with increasing size of the query window.

Next, we investigate the scalability of algorithms with respect to the size of the dataset. Figure 5 reports processing, storage and preparation costs for the algorithms with respect to the size of the dataset. The datasets for this experiment are obtained by selecting subsets of predetermined sizes from full datasets. We fix the size of the query window to 1% of the time domain and we set the clustering proportion to 0.9. The result demonstrates that STI scales better than its competitors with respect to the dataset size in all of the measured metrics. This result demonstrates that given the same budget (for index preparation and storage), STI will be significantly more effective than other methods in query processing in both small and large datasets.

Note that the processing cost of STI changes little as the size of query window increases. This clearly demonstrates that the processing cost within the query window is not the efficiency bottleneck for STI approach. In other words, the scanning cost in the living history window takes up the most time in STI's processing. In the rest of this section, we would present how various checkpointing (CP) strategies could reduce this cost and further improve the effectiveness of the STI approach.

Investigation for STI-CP. Figure 6 reports the processing cost of several checkpointing strategies (STI-CPs) with respect to the size of the query window. We set the budget parameter B to 0.6% and clustering proportion to 0.9. We note that the query-set strategy outperforms all other strategies in processing short queries and its advantage diminishes as the size of the query window increases. This is expected because checkpointing strategies aim at reducing the computation cost within the living history window.

In following experiments, we investigate the effectiveness of STI-CPs with respect to the size of the dataset, given checkpoint budget, and the distribution of queries in the workload. We use *instant* queries in these experiments since this isolates the effect of the size of the living history window on checkpointing strategies.

Figure 7 reports variation in query processing cost for STI-CPs with respect to the size of the dataset. We note that the three data-aware strategies perform close to each other but all outperform the random strategy. The most important result is that for all STI-CPs, the query-set strategy outperforms the data-aware strategies in both small and large datasets.

We also record the time consumption on checkpoint-selecting of various STI-CPs and the result demonstrates that the query-set strategy needs more time to select the checkpoints. However, this cost is in the magnitude of milliseconds, which is a very small part of the total preparation cost of the STI index are could be negligible. So they are not reported in figures.

Figure 8 reports the performance of STI-CPs with respect to the checkpoint budget B. In most situations, the query-set strategy outperforms other strategies in processing time. The only exception is in FHV with $B = 0.2\%$ where it performs similar to the others. This is expected as average CSS of checkpoints in FHV is much larger than the other datasets, so query-set cannot set enough checkpoints for all clusters when the budget is low. As the budget increases, the advantage of the query-set strategy becomes apparent.

Figure 9 reports the performance of STI-CPs with respect to the distribution of queries in a workload. Observe that the query-set strategy performs best when the clustering ratio r is in [0.8, 0.95]. However, the advantage of the query-set strategy declines when $r = 0.5$ and $r = 1$. Query-set checkpointing strategy does not perform as well when $r = 0.5$ due to lack of clustered queries, hence limited query-aware optimization is possible. In other words, the checkpoints for clusters do not have much influence on overall processing cost. Query-set performance when $r = 1$ can be explained by the cluster-detecting method we use. By analyzing the details in

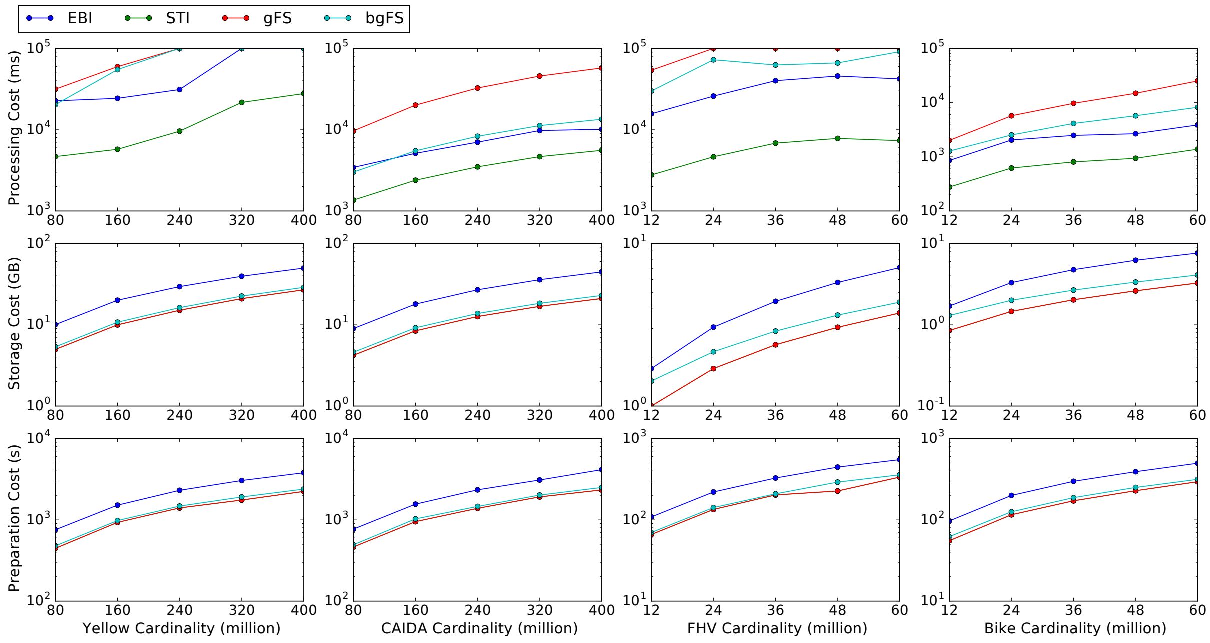

Figure 5: Performance of basic algorithms with respect to the dataset size.

checkpoint selection procedure, we find that mean-ISI method cannot properly identify the duration of each cluster when there are no uniform queries in workload. That is, the identified duration does not completely cover all queries in a cluster yet it could identify the correct number of clusters. This is due to the threshold which is used for clustering being smaller than the possible maximum inter-time between two consecutive *qstart* in the same cluster, when queries are completely clustered. However, when some uniform queries are introduced (i.e., $r < 1$), the calculated threshold could be lifted so it can not properly cover cluster duration and filter uniform queries.

To further understand the effect of the chosen clustering method, we carry out an additional experiment to test the performance of the query-set strategy with respect to the size of training set, reported in Figure 10. Comparing to the processing cost of other strategies (when $x = 0\%$ in Figure 6), we observe that the query-set one begins to outperform the others when training set increases to 0.4 of the test set, and its advantage becomes more stable and apparent when the ratio increases to 0.6-1.0. This is expected because test set used in experiments are small (1000 queries in each) so the smaller training set is not enough for the query-set strategy to learn the clustering structures especially when training set ratio is low. In other words, if we increase the size of processing set, even the training set at the size of 0.2 of processing set could present clear clustering structure information for the query-set strategy.

7 CONCLUDING REMARKS

In this paper, we propose STI and STI-CP approaches for temporal k-clique enumeration. STI is designed to overcome the efficiency bottlenecks in the state-of-art methods and STI-CP use checkpoints to further improve processing efficiency. Our experimental results

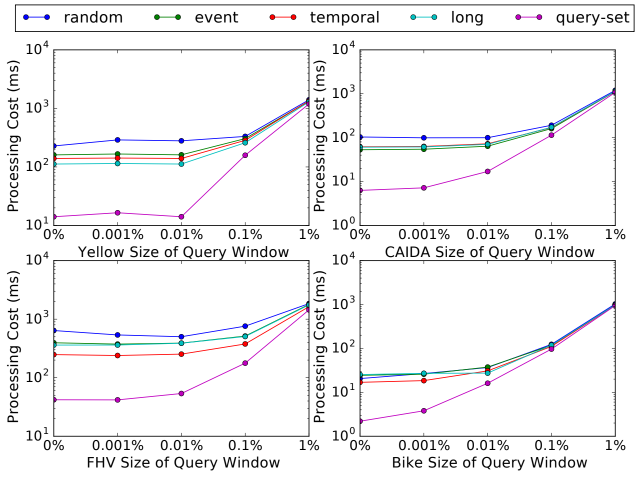

Figure 6: Performance of STI-CPs with respect to the size of the query window.

demonstrate that STI outperforms current state-of-the-art methods and all proposed checkpointing strategies outperform random checkpoint selection method by a wide margin. In future work, we plan to study additional checkpointing strategies and the applicability of STI and STI-CP approaches in the context of scalable temporal graph analytics methods.

8 ACKNOWLEDGEMENTS

This work was partially funded by the EU H2020 project Smart-DataLake (825041).

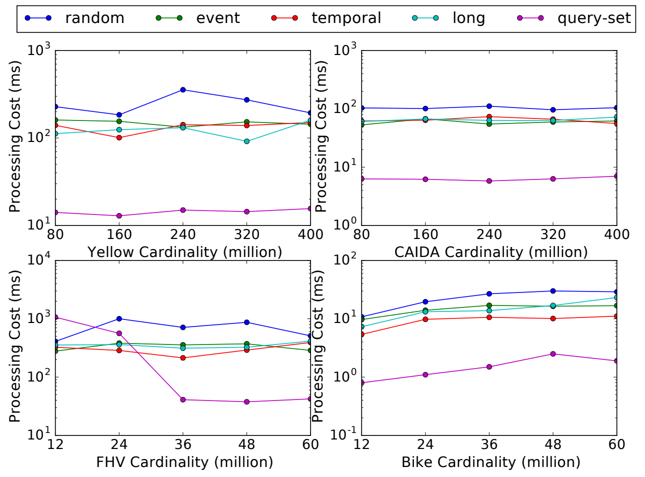

Figure 7: Performance of STI-CPs with respect to the dataset size.

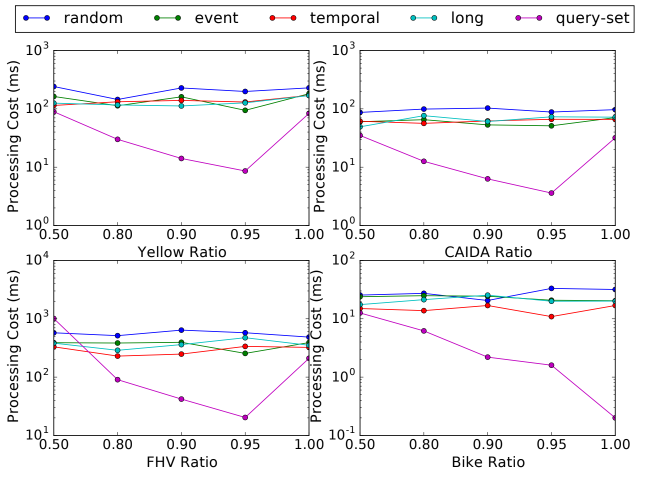

Figure 9: Performance for STI-CPs in varying query ratio.

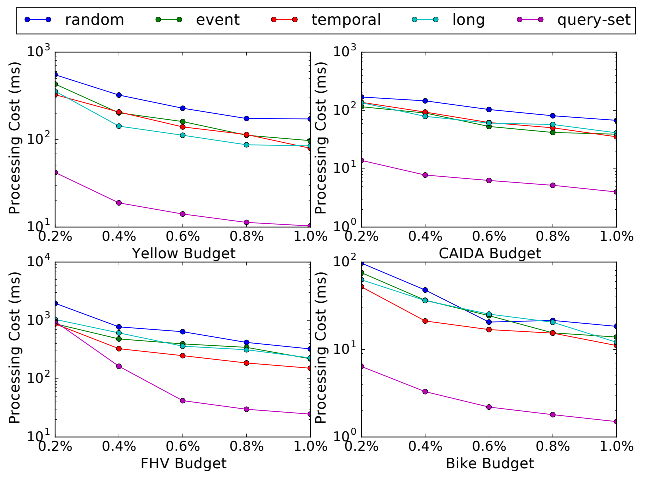

Figure 8: Performance for STI-CPs with respect to the budget

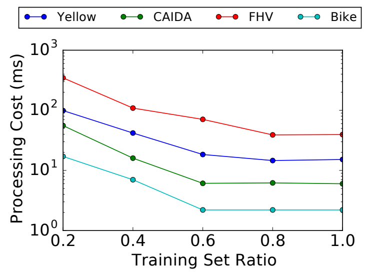

Figure 10: Performance for query-set STI-CP with respect to the size of the training set

REFERENCES

[1] CAIDA UCSD anonymized internet traces dataset - [dates used]. http://www.caida.org/data/passive/passive_dataset.xml.
[2] NYC citi-bikes. https://www.citibikenyc.com/system-data.
[3] NYC Free Hired Vehicles. http://www.nyc.gov/html/tlc/downloads/pdf/data_dictionary_trip_records_fhv.pdf.
[4] NYC Yellow Taxi. http://www.nyc.gov/html/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf.
[5] Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proceedings of the VLDB Endowment*, 10(11):1346–1357, 2017.
[6] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. *Efficient processing of spatial joins using R-trees*, volume 22. ACM, 1993.
[7] Francesco Cafagna and Michael H Böhlen. Disjoint interval partitioning. *The VLDB Journal*, 26(3):447–466, 2017.
[8] Lin Chen, Yong Deng, Weihua Luo, Zhen Wang, and Shaoqun Zeng. Detection of bursts in neuronal spike trains by the mean inter-spike interval method. *Progress in Natural Science*, 19(2):229–235, 2009.
[9] Reynold Cheng, Sarvjeet Singh, Sunil Prabhakar, Rahul Shah, Jeffrey Scott Vitter, and Yuni Xia. Efficient join processing over uncertain data. In *International Conference on Information and Knowledge Management CIKM*, pages 738–747, 2006.
[10] Anton Dignös, Michael H Böhlen, and Johann Gamper. Overlap interval partition join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1459–1470. ACM, 2014.
[11] Jost Enderle. Joining interval data in relational databases. In *ACM SIGMOD International Conference on Management of Data*, pages 683–694, 2004.
[12] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, 2005.
[13] K-I Goh and A-L Barabási. Burstiness and memory in complex systems. *EPL (Europhysics Letters)*, 81(4):48002, 2008.
[14] Martin Kaufmann, Peter M Fischer, Norman May, Chang Ge, Anil K Goel, and Donald Kossmann. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *IEEE International Conference on Data Engineering*, pages 471–482, 2015.
[15] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: a unified data structure for processing queries on temporal data in sap hana. In *ACM SIGMOD International Conference on Management of Data*, pages 1173–1184. ACM, 2013.
[16] Ji-Zhou Luo, Sheng-Fei Shi, Guang Yang, Hong-Zhi Wang, and Jian-Zhong Li. O2ijoin: An efficient index-based algorithm for overlap interval join. *Journal of Computer Science and Technology*, 33(5):1023–1038, 2018.
[17] Burçak Otlu and Tolga Can. Joa: Joint overlap analysis of multiple genomic interval sets. *BMC bioinformatics*, 20(1):121, 2019.
[18] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1098–1109. IEEE, 2016.