Distributed Evaluation of Top-k Temporal Joins

Julien Pilourdault, Vincent Leroy, Sihem Amer-Yahia

Distributed Evaluation of Top-k Temporal Joins

Julien Pilourdault, Vincent Leroy, Sihem Amer-Yahia
Université Grenoble Alpes - LIG, CNRS
Grenoble, France
Firstname.Lastname@imag.fr

ABSTRACT

We study a particular kind of join, coined Ranked Temporal Join (RTJ), featuring predicates that compare time intervals and a scoring function associated with each predicate to quantify how well it is satisfied. RTJ queries are prevalent in a variety of applications such as network traffic monitoring, task scheduling, and tweet analysis. RTJ queries are often best interpreted as top-k queries where only the best matches are returned. We show how to exploit the nature of temporal predicates and the properties of their associated scoring semantics to design *TKIJ*, an efficient query evaluation approach on a distributed Map-Reduce architecture. *TKIJ* relies on an offline statistics computation that, given a time partitioning into granules, computes the distribution of intervals' endpoints in each granule, and an online computation that generates query-dependent score bounds. Those statistics are used for workload assignment to reducers. This aims at reducing data replication, to limit I/O cost. Additionally, high-scoring results are distributed evenly to enable each reducer to prune unnecessary results. Our extensive experiments on synthetic and real datasets show that *TKIJ* outperforms state-of-the-art competitors and provides very good performance for n-ary RTJ queries on temporal data.

Keywords

Temporal data; Top-k; Join; Distributed processing

CCS Concepts

•**Information systems** → **Join algorithms;** *Query operators;*

1. INTRODUCTION

Temporal data is pervasive. Store receipts, tweets, traffic data, and temperature measures generated by weather sensors or by wearables, are just some examples. Such data is best represented as intervals with start and end timestamps. For instance, in a data center, large amounts of traffic requests in the form of [*IP_address*,*start timestamp*,*end timestamp*,...], are continuously generated by sensors. A system administrator who wishes to monitor traffic emanating from different countries, would formulate a query that returns pairs of requests (x, y) where x ends *before* y starts and x and y originate from different countries. In tweet analysis where intervals represent the lifespan of a hashtag, the query x *meets* y would find pairs of discussion topics where one started when the other ended.

Chronological relations between intervals are typically expressed using the Allen interval algebra [2], which defines a set of Boolean predicates such as *before, meets, starts* and *overlaps*. In this work, we argue that the ability to evaluate interval predicates *approximately* and assign scores to resulting interval pairs, appears as a natural requirement to finding interesting results. Indeed, in monitoring traffic causality, an (x, y) pair where x ends *just before* y, may be preferred to those where x ends *much earlier than* the start of y. Similarly, in tweet analysis, a Boolean interpretation of x *meets* y is likely to return an empty result and will not detect discussion topics that started *roughly* after another ended.

In this paper, we formalize n-ary Ranked Temporal Join (RTJ) queries that feature any number of temporal predicates and return the k best results. To the best of our knowledge, this is the first work that formalizes n-ary RTJ queries and devises a general-purpose distributed evaluation approach. RTJ queries raise a number of new challenges. First, an appropriate ranked semantics needs to be devised for a variety of temporal predicates in order to capture the strength of relationship between intervals as a function of the desired semantics for each predicate. Most related studies focus on a common query that involves the *intersects* predicate [7, 13, 21]. The idea is to retrieve tuples that share a common period of validity, in order to combine events whose time range has a non-empty intersection. Our semantics is richer since we are interested in any temporal predicate between intervals. For instance, we aim to capture predicates from the Allen algebra (see the three first columns of Figure 2). We illustrate that in the following example.

Network Traffic Monitoring. Consider the two interval collections C_1 and C_2 in Figure 1. Assume that each collection gathers requests from a different country. An analyst interested in monitoring traffic between two countries would seek (x, y) interval pairs, $x \in C_1$, $y \in C_2$, that form a sequence, i.e., where y starts as x ends. The best sequences are those satisfying $meets(x, y)$ reflecting strict equality between the end of x and the beginning of y. In our example, only

To appear in SIGMOD'16

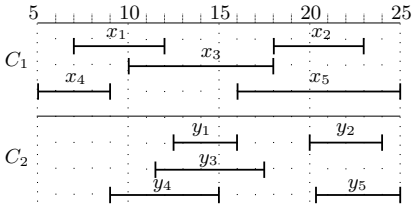


Figure 1: Motivating Example

(x_4, y_4) qualifies for a Boolean interpretation. Yet, given the uncertainty on clocks in different network equipments, a small overlap or a short free period between two consecutive tasks would be more realistic. In order to express that with Boolean semantics, a query must evaluate a disjunction of *before*, *overlaps* and *meets* and would build many useless interval pairs. A ranked semantics on the other hand, would build (x, y) pairs where x *almost meets* y, allowing a tolerance on intervals' endpoints. For instance, tuples (x, y) where x ends at most 2 timestamps before or after y starts, could be considered high-scoring. Using such a semantics, we can return the top-3 $\{(x_4, y_4), (x_1, y_3), (x_1, y_1)\}$.

A key observation we rely on is that an approximate interpretation of a predicate amounts to approximating the strength of relationship between its intervals' endpoints. That is compatible with the flexible scoring approach proposed in [12] to approximate Allen predicates [2]. It is based on associating a degree of satisfaction with equality and inequality of 2 intervals' endpoints. In this paper, we adapt this framework to allow scoring any temporal predicate. We also allow to use any monotone aggregation function to compute the score of a query result for n-ary RTJ queries. Consequently, a 3-way query involving the predicates $starts(x, y)$ and $meets(y, z)$ would return (x, y, z) tuples and associate to each tuple its degree of satisfaction of the query as an aggregation of individual predicate-dependent scores for each of $starts(x, y)$ and $meets(y, z)$.

It is important to note that, unlike existing work on returning approximate answers [4], we focus on computing exact answers of queries that make use of scored join predicates. The second challenge we tackle is to devise an efficient query evaluation strategy that guarantees to return the best answers for a variety of temporal predicates. The key difficulty here is to develop a general-purpose algorithm that works with a variety of predicates and ranked semantics and yet, that is able to exploit the nature of those predicates to devise an efficient evaluation.

The efficient processing of interval joins has been studied before [5, 7, 13, 21]. The closest to our work is the recent one by Chawda et al. [5] with a focus on processing queries on Map-Reduce [6]. However, their algorithms focus on a Boolean semantics and are not directly applicable in our case. In our work, scores constitute both a challenge and an opportunity. They are a challenge because, unlike Boolean semantics, every combination of intervals is potentially an answer. The opportunity lies in the ability to leverage statistics on input data in order to avoid computing low-scoring results. In this paper, we propose *TKIJ*, a 3-stage query evaluation approach for RTJ queries on Map-Reduce. A first offline process partitions time into granules and computes the distribution of intervals' endpoints in those granules. A second online process leverages those statistics to compute query-dependent score bounds. A third process assigns to each reducer enough high-scoring results to enable pruning unnecessary answers and hence balance the load between reducers. Each reducer executes the input RTJ query locally on the data it receives. While this query processing bears similarities to traditional *top-k* and *rank-join* processing [14, 15, 16, 20, 22], it differs greatly due to the dependence of score computation on query predicates. As a result, partial predicate-dependent scores cannot be pre-computed and each reducer relies on the score bounds produced in the statistics collection phase to avoid unnecessary computation. A final Map-Reduce job merges results obtained from each reducer.

In summary, this paper makes the following contributions:

1. We formalize n-ary RTJ queries that combine interval collections with any number of arbitrary temporal predicates and return the k best results.

2. We design *TKIJ*, an efficient query evaluation approach on Map-Reduce that leverages scores to avoid computing unnecessary results and to balance the load between reducers.

3. We run extensive experiments on synthetic and real network traffic datasets. Our experiments show the efficiency of our pruning technique and the great benefit of using scores to distribute the load between reducers. Because *TKIJ* executes only combinations that ensure to return the correct top-k answers, it scales to very large collections (up to 5M tuples per collection) and to high k values (up to 10^5).

Section 2 describes our data model and defines the problem of evaluating RTJ queries. *TKIJ* is presented in Section 3. Experiments are detailed in Section 4. We summarize the related work in Section 5 and conclude in Section 6.

2. DATA MODEL AND PROBLEM

We are given m collections of intervals $C_1, \ldots, C_m$. Each interval x has a unique identifier, a start time $\underline{x}$ and an end time $\overline{x}$.

Boolean temporal predicates. The general form of a temporal predicate between two intervals x and y is denoted $p(x, y)$ and is expressed as a Boolean conjunction of equalities and inequalities between their endpoints $\underline{x}, \overline{x}, \underline{y}, \overline{y}$. This allows to capture a wide range of predicates among which the seminal Allen algebra [2]. The first 3 columns of Figure 2 summarize Allen temporal predicates and their semantics. For example, $meets(x, y)$ imposes that y starts when x finishes while $starts(x, y)$ requires that x and y start at the same time and that x ends before y.

It is important to note that we aim to capture all Allen predicates but also any predicate comparing interval's endpoints. While we do not aim to provide a long list of new predicates, we discuss examples that we are using in our experiments. For example, in network traffic analysis, we introduce $justBefore(x, y)$ that is satisfied iff $\underline{y} > \overline{x}$ and $\underline{y} - \overline{x} \leq AVG_z(\overline{z} - \underline{z})$. The intuition is that the elapsed time between x and y is no greater than the average interval length. A special case is the predicate $shiftMeets(x, y)$ that is satisfied iff $(\underline{y} - \overline{x}) = AVG_z(\overline{z} - \underline{z})$. In tweet analysis, a possibly useful predicate would be $sparks(x, y)$ which

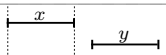
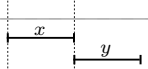
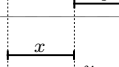
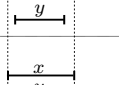
Temporal Predicate	Boolean Interpretation	Valid answers	Scored Intepretation
$before(x,y)$	$\overline{x} < \underline{y}$		***s-before***$(x,y) =$ $greater(\underline{y},\overline{x})$
$equals(x,y)$	$\underline{x} = \underline{y} \wedge \overline{x} = \overline{y}$		***s-equals***$(x,y) =$ $\min\{equals(\underline{x},\underline{y}),$ $equals(\overline{x},\overline{y})\}$
$meets(x,y)$	$\overline{x} = \underline{y}$		***s-meets***$(x,y) =$ $equals(\overline{x},\underline{y})$
$overlaps(x,y)$	$\underline{x} < \underline{y} \wedge \overline{x} > \underline{y}$ $\wedge\ \overline{x} < \overline{y}$		***s-overlaps***$(x,y) =$ $\min\{greater(\underline{y},\underline{x}),$ $greater(\overline{x},\underline{y}),$ $greater(\overline{y},\overline{x})\}$
$contains(x,y)$	$\underline{x} < \underline{y} \wedge \overline{x} > \overline{y}$		***s-contains***$(x,y) =$ $\min\{greater(\underline{y},\underline{x}),$ $greater(\overline{x},\overline{y})\}$
$starts(x,y)$	$\underline{x} = \underline{y} \wedge \overline{x} < \overline{y}$		***s-starts***$(x,y) =$ $\min\{equals(\underline{x},\underline{y}),$ $greater(\overline{y},\overline{x})\}$
$finishedBy(x,y)$	$\underline{x} < \underline{y} \wedge \overline{x} = \overline{y}$		***s-finishedBy***$(x,y) =$ $\min\{greater(\underline{y},\underline{x}),$ $equals(\overline{x},\overline{y})\}$

Figure 2: The Allen Algebra with Boolean and Scored Temporal Predicates.

is satisfied iff $(\overline{y} - \underline{y}) > 10 * (\overline{x} - \underline{x})$ and $\underline{y} > \overline{x}$. As a result, the (x, y) pairs satisfying $sparks(x, y)$ would identify hashtag pairs where the preceding hashtag lasted 10 times shorter that the following. That could be useful in determining causality of long-lasting events such as finding all short-lasting hashtags before the long-lasting #`JeSuisCharlie`.

Scored temporal predicates. Since we are interested in capturing the degree at which a temporal predicate is verified by a pair of intervals, we propose to associate a score to each predicate. Here again, we aim to be general and we adopt the flexible approach for scoring Allen predicates [12] and adapt it to our settings. This approach relies on two primitive approximation comparators on intervals' endpoints. Those comparators, $equals(\overline{x},\overline{y})$ and $greater(\overline{x},\overline{y})$, are used to express the degree of equality or inequality of intervals' endpoints $\overline{x}$ and $\overline{y}$, where $\overline{x} \in \{\underline{x},\overline{x}\}$, $\overline{y} \in \{\underline{y},\overline{y}\}$ as a graded value in $[0, 1]$. They rely on two parameters λ and ρ that provide flexibility in controlling the tolerance degree when comparing intervals' endpoints. Figure 3 shows how $equals(\overline{x},\overline{y})$ and $greater(\overline{x},\overline{y})$ are used with λ and ρ to express that tolerance. For instance, by defining that whenever $|\overline{x} - \overline{y}| \leq \lambda$, $equals(\overline{x},\overline{y})$ returns 1, λ sets a tolerance for exact endpoint equality. ρ, on the other hand, is used to define score values. A large ρ value defines a wide range of score values and a small ρ produces a more abrupt curve and fewer possible score values.

Since temporal predicates are expressed as equalities and inequalities on intervals' endpoints, their approximation can be achieved using a conjunction of *equals*() and *greater*() with appropriate λ and ρ values. This allows us to *associate a scored variant to each temporal predicate*. We denote that variant *s-p*(x, y) and refer to it as *scored temporal predicate*, abusing the term "predicate" to mean "function". Indeed, while $p(x, y)$ returns a Boolean value, *s-p*(x, y), returns a score in [0,1]. For example, we can define the scored version of $starts(x, y)$ as *s-starts*$(x, y) = \min\{equals(\underline{x},\underline{y}),$ $greater(\overline{y},\overline{x})\}$.

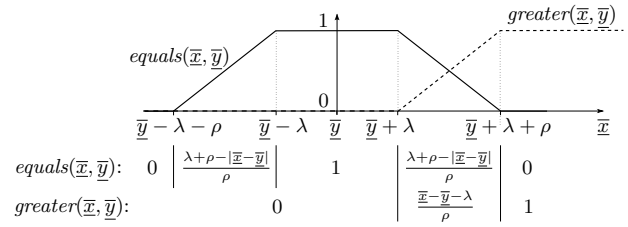


Figure 3: Approximating *equals* and *greater*. Here, $\overline{x} \in \{\underline{x},\overline{x}\}$, $\overline{y} \in \{\underline{y},\overline{y}\}$.

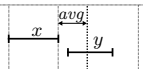
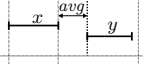
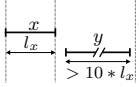
Temporal Predicate	Boolean Interpretation	Valid Answers	Scored Intepretation
$justBefore(x,y)$	$\overline{x} < \underline{y} \wedge$ $\underline{y} - \overline{x} \leq avg$		***s-justBefore***$(x,y) =$ $\min\{equals(\overline{x},\underline{y}),$ $greater(\underline{y},\overline{x})\}$ $\lambda_{greater} = \rho_{greater} = 0,$ $\lambda_{equals} = avg,\ \rho_{equals} \in \mathbb{R}^+$
$shiftMeets(x,y)$	$\underline{y} = \overline{x} + avg$		***s-shiftMeets***$(x,y) =$ $equals(\overline{x} + avg, \underline{y})$
$sparks(x,y)$	$\overline{x} < \underline{y} \wedge$ $(\overline{y} - \underline{y}) >$ $10 * (\overline{x} - \underline{x})$		***s-sparks***$(x,y) =$ $\min\{greater(\underline{y},\overline{x}),$ $greater(\overline{y} - \underline{y}, 10 * (\overline{x} - \underline{x}))\}$

Figure 4: Definition of *s-shiftMeets*, *s-justBefore* and *s-sparks*. Here, $avg = AVG_z(\overline{z} - \underline{z})$.

We also propose to allow different values of λ and ρ for *equals*() and *greater*() for different predicates. That provides a finer control of the score values produced by each predicate. A Boolean interpretation of a predicate becomes a special case of our scored interpretation. For example, strict endpoint equality can be obtained by setting both λ_{equals} and ρ_{equals} to 0. Thus, we can define, *s-justBefore*(x, y) with $\lambda_{greater}$ and $\rho_{greater}$ set to 0, ρ_{equals} to any value and λ_{equals} to $AVG_z(\overline{z} - \underline{z})$ (Figure 4).

Temporal join queries. We are interested in expressing n-ary join queries on interval collections $C_1, \ldots, C_m$. We express a query Q as a weakly connected oriented simple graph[1] of the form (V, E). Each each vertex $v_i \in V$ is mapped to a collection C_i. Each edge $(i, j) \in E$ between two vertices v_i and v_j is labeled with a scored temporal predicate *s-*$p_{(i,j)}()$ between the two collections C_i and C_j corresponding to v_i and v_j.

The evaluation of an n-ary join query Q returns a set of tuples of the form $(x_1, \ldots, x_n)$ where $x_i \in C_i$. The score of each tuple in the query result is computed using a function $\mathcal{S}$ that aggregates the partial scores assigned by each predicate *s-*$p_{(i,j)}()$ associated with each query edge $(i, j) \in E$. $\mathcal{S}$ could be any monotone function such as the weighted sum as it is commonly the case in ranked aggregation [15, 16, 20, 22].

For example, we can express a 3-way query that returns a tuple (x, y, z) where $x \in C_1$, $y \in C_2$ and $z \in C_3$ and the score of (x, y, z) is computed as an aggregation of its partial scores for query predicates *s-starts*(x, y) and *s-meets*(y, z).

Although we use the term "join" to refer to our queries, their expressivity goes beyond traditional relational joins. Our queries are compositional in the sense of a relational join

[1]no self loops and $(i, j) \in E \implies (j, i) \notin E$

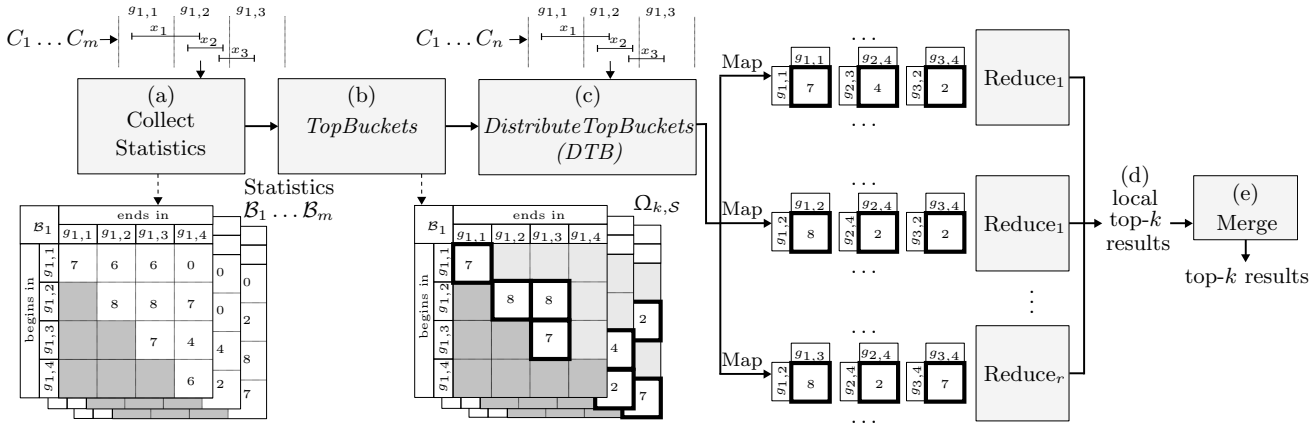


Figure 5: Overview of *TKIJ*

since their results are not intervals but tuples of any length (corresponding to the number of vertices in the query). Our queries can express any combination of interval collections with any scored predicates including chain queries and queries containing cycles.

Ranked Temporal Join (RTJ) problem. Given an n-ary temporal join query $Q = (V, E)$ expressed over a set of collections $C_1, \dots, C_m$ corresponding to query vertices in V and temporal predicates $s\text{-}p_{(i,j)}()$ associated to each edge $(i, j) \in E$, our problem is to find a top-k set of tuples of the form $(x_1, \dots, x_n)$, $x_i \in C_i$, ranked by (descending) order of $\mathcal{S}_{(i,j)\in E}(s\text{-}p_{(i,j)}(x_i, x_j))$.

3. TEMPORAL JOIN PROCESSING

We present *TKIJ*, our approach for evaluating Top-K Interval Joins, that efficiently finds the set of k *best* results for an RTJ query Q. We first provide an overview of *TKIJ*, then we give each step in detail.

3.1 Overview of TKIJ

Figure 5 summarizes *TKIJ*. Given a set of interval collections $C_1 \dots C_m$, *TKIJ* executes a query-independent preprocessing phase to collect statistics on intervals' distribution. This phase partitions time into granules and computes *buckets* for each collection (a). A bucket associated to a collection C_i corresponds to a pair of granules, and contains the number of intervals of C_i starting at one granule and ending at another. Given a query Q, these statistics are used to evaluate *bucket combinations* that should be processed in order to obtain top-k results (b). *TKIJ* relies on a constraint programming solver to compute score bounds for each bucket combination and uses those bounds to prune combinations that do not contain top-k results. The third phase is the actual join processing which relies on two Map-Reduce jobs. The first job assigns a subset of buckets to each reducer r_j (c) which then processes locally the RTJ query, returning local top-k results (d). This assignment aims at reducing data replication to limit I/O, and leverages score bounds to distribute high-scoring results evenly so that each reducer can quickly prune low-ranking results. The second Map-Reduce job merges all local results into a single query output (e).

3.2 Statistics collection

TKIJ pre-processes each dataset once in order to collect statistics which are then used to optimize the execution of any RTJ query on this dataset. These statistics maintain a matrix $\mathcal{B}_i$ representing the distribution of endpoints of intervals in each collection C_i. *TKIJ* partitions the time range of each C_i into a set of contiguous *granules*. We adopt a uniform partitioning which has been shown to be appropriate for temporal joins [5, 7, 18].

As illustrated in Figure 5a, each matrix entry records the cardinality of a bucket, where a bucket $b_{i,l,l'} = (g_{i,l}, g_{i,l'})$ contains all intervals of C_i that start in $g_{i,l}$ and end in $g_{i,l'}$: $\mathcal{B}_i[l][l'] = |b_{i,l,l'}| = |\{x \in C_i, \underline{x} \in g_{i,l} \wedge \overline{x} \in g_{i,l'}\}|$. As an example, given $g_{1,1} = [10, 20]$ and $g_{1,2} = [20, 30]$, the matrix entry for $b_{1,1,2} = (g_{1,1}, g_{1,2})$ indicates 6 intervals starting in $[10, 20]$ and ending in $[20, 30]$.

Range partitioning is a common approach in temporal join processing [5, 7, 18, 21]. The rationale is that intervals having similar endpoints are likely to satisfy similar join predicates. For example, most previous studies, that focus on *intersection* joins, leverage partitions to avoid pairs of intervals that are guaranteed not to intersect. Similarly, *TKIJ* relies on these statistics to obtain information on the distribution of intervals within buckets and prune the search space of *any* RTJ query.

Statistics are computed in a single Map-Reduce phase. Each mapper reads a fraction of the data and maintains a local matrix per collection. Matrices are then aggregated in the reduce phase, and the reducer responsible for collection C_i outputs a final matrix $\mathcal{B}_i$. While we focus in this paper on the case of statistics computed from scratch for a new dataset, we can easily handle updates by applying the same process on the inserted/deleted data.

3.3 Selection of bucket combinations

We now describe how *TKIJ* uses pre-computed statistics to estimate score bounds on candidate results. Then, we present how score bounds are used to avoid computing unnecessary results while we guarantee to return the exact top-k results. We further develop several processing strategies that aim to tackle computational costs raised by this pruning step.

Estimating score bounds

Processing an RTJ query Q requires to return the top-k tuples $(x_1, \ldots, x_n)$, $x_i \in C_i$ according to a scoring function $\mathcal{S}$. Since any tuple $(x_1, \ldots, x_n)$ is a potential answer, we investigate how to reduce the amount of data processed using scores. We use $\omega = (b_{1,l_1,l'_1}, \ldots, b_{n,l_n,l'_n})$ to denote a bucket combination, $\omega.nbRes = \prod_{i=1}^{n} |b_{i,l_i,l'_i}|$ the total number of results that can be obtained from a bucket combination ω, and Ω the set of all combinations. We define score upper and lower bounds in each ω, denoted $\omega.UB$ and $\omega.LB$.

Definition 1. The score upper-bound (resp. lower-bound) $\omega.UB$ (resp. $\omega.LB$) of a bucket combination $\omega = (b_{1,l_1,l'_1}, \ldots, b_{n,l_n,l'_n})$ is the upper-bound (resp. lower-bound) of $\mathcal{S}_{(i,j)\in E}(s\text{-}p_{(i,j)}(x_i, x_j))$ where $\underline{x_i} \in g_{i,l_i}, \overline{x_i} \in g_{i,l'_i}, \forall i \in 1 \ldots n$.

As an example, suppose that query Q features a predicate $s\text{-}meets_{(1,2)}(x, y)$ where $x \in C_1$ and $y \in C_2$, using scoring parameters $(\lambda_{equals}, \rho_{equals}) = (4, 8)$. Collected statistics show 6 intervals in bucket $b_{1,1,2} = ([10, 20], [20, 30])$ for C_1 and 7 intervals in bucket $b_{2,2,3} = ([20, 30], [30, 40])$ for C_2. We build the bucket combination $\omega = (b_{1,1,2}, b_{2,2,3})$. Then, we can derive bounds on the score $\mathcal{S}(x, y) = s\text{-}meets(x, y)$ of a result $(x, y) \in \omega$. The maximum possible score is 1 (e.g. with $(x, y) = ([12, 25], [25, 35])$), and the minimum score is 0.25 (with $(x, y) = ([15, 20], [30, 35])$). Hence, $\omega.UB = 1, \omega.LB = 0.25$. Thus, 42 results in ω have a score in $[0.25, 1]$.

TKIJ relies on a constraint programming solver as a generic approach to compute score bounds for any combination of predicates. Computing score bounds for a bucket combination requires to solve the following problem:

Bounds Problem. *Let $\omega = (b_{1,l_1,l'_1}, \ldots, b_{n,l_n,l'_n})$. Find $(x_1, \ldots, x_n)$ s.t.:*

$$\begin{aligned} \textit{max (resp. min)} \quad & \mathcal{S}_{(i,j)\in E}(s\text{-}p_{(i,j)}(x_i, x_j)) \\ \textit{s.t.} \quad & \underline{x_i} \in g_{i,l_i} \ \forall i \in 1 \ldots n \qquad (1) \\ & \overline{x}_i \in g_{i,l'_i} \ \forall i \in 1 \ldots n \qquad (2) \end{aligned}$$

Each $x_i \in C_i$ is mapped to a decision variable $\mathtt{x}_i$. For each partial score $s\text{-}p_{(i,j)}(x_i, x_j)$, we create an intermediate variable $\mathtt{s\text{-}p_{ij}}$. For all $(i, j) \in E$, we impose that the variables $\mathtt{x}_i, \mathtt{x}_j$ and $\mathtt{s\text{-}p_{ij}}$ satisfy the constraints $\{\mathtt{C}_{ij} : \mathtt{s\text{-}p_{ij}} = s\text{-}p_{(i,j)}(\mathtt{x}_i, \mathtt{x}_j)\}$. Then, we create a variable `score` and impose $\mathtt{C}_s : \mathtt{score} = \mathcal{S}_{(i,j)\in E}(\mathtt{s\text{-}p_{ij}})$. The solver then maximizes (and minimizes in the case of a lower-bound) `score` such that constraint $\mathtt{C}_s$, all constraints $\mathtt{C}_i$ and all constraints on decision variables (1)(2) are satisfied. While we virtually allow any temporal predicates, in practice, predicate implementation depends on the range of constraints supported by the server used in the implementation.

Pruning bucket combinations

TKIJ leverages computed score bounds to reduce computation cost by eliminating results that are guaranteed not to be in the top-k. To do so, it computes $\Omega_{k,\mathcal{S}} \subseteq \Omega$, a subset of the search space that is sufficient to guarantee correctness. We define $\Omega_{k,\mathcal{S}}$ as follows:

Definition 2. The set of Top Buckets $\Omega_{k,\mathcal{S}}$ is a subset of Ω satisfying the following conditions:

- $\forall \omega \in \Omega \setminus \Omega_{k,\mathcal{S}} \ \exists \Psi \subseteq \Omega_{k,\mathcal{S}}$:
  - $\forall \omega' \in \Psi \ \omega'.LB \geq \omega.UB$
  - $\sum_{\omega' \in \Psi} \omega'.nbRes \geq k$

Algorithm 1 *getTopBuckets*

Input: k, Ω list of bucket combinations with UB and LB
Output: $\Omega_{k,\mathcal{S}}$

```
1: Sort Ω by descending LB
2: collectedResults = 0
3: for ω ∈ Ω
4:     collectedResults += ω.nbRes
5:     kthResLB = ω.LB
6:     if collectedResults ≥ k then break
7: Sort Ω by descending UB
8: Ω_{k,S} ← ∅, collectedResults = 0
9: for ω ∈ Ω
10:    if collectedResults ≥ k and ω.UB ≤ kthResLB
11:        break
12:    Ω_{k,S} ← Ω_{k,S} ∪ ω
13:    collectedResults += ω.nbRes
   return Ω_{k,S}
```

This definition ensures that whenever a bucket combination ω is pruned, there are at least k results from $\Omega_{k,\mathcal{S}}$ with a score higher than results generated from ω. Note that $\Omega_{k,\mathcal{S}}$ is not unique: given a valid set of bucket combinations, any super-set is also valid. To compute $\Omega_{k,\mathcal{S}}$, we design the *getTopBuckets* algorithm (Algorithm 1). Algorithm *getTopBuckets* uses as input a set of bucket combinations whose score bounds are pre-computed. It first computes a lower bound *kthResLB* on the score of the k^{th} result (Lines 1-6). Then, it keeps only bucket combinations whose score upper-bound is greater than *kthResLB* (Lines 7-13). The process safely stops in Line 11 since no bucket combination outside the collected set has results with score above *kthResLB*.

Pruning unnecessary results is a two-step process, coined *TopBuckets*. A first step computes score bounds for bucket combinations using a solver. Then, a second step executes *getTopBuckets* that uses those bounds to eliminate unnecessary results. In our setting, all $(x_1, \ldots, x_n)$ combinations are potential answers, and we cannot employ traditional top-k techniques to prune the search space. *TopBuckets* addresses this challenge using pre-computed statistics to locate high-scoring answers. Still, a new challenge is to limit the overhead of *TopBuckets* due to computing score bounds for bucket combinations. In the following section, we discuss this challenge and possible solutions.

TopBuckets Strategies

A first straightforward approach to find $\Omega_{k,\mathcal{S}}$ is to build all possible bucket combinations Ω, compute their score bounds using a solver and then use *getTopBuckets* to prune useless ones. In this strategy, coined BRUTE-FORCE, a large number of n-tuples of buckets are assigned a score by the solver (with g granules per collection, $|\Omega|$ is $\mathcal{O}(g^{2n})$). Moreover, for each combination, $2n$ decision variables need to be assigned by the solver. Thus, as n or g increase, BRUTE-FORCE becomes inefficient.

To tackle that, we propose the LOOSE strategy (Algorithm 2 with the flag `onePhase` set `true`.) LOOSE first builds all bucket pairs $(b_{i,l_i,l'_i}, b_{j,l_j,l'_j})$ for each scored predicate $(i, j) \in E$ (Line 1). Score bounds are then computed by the solver (Line 3) for each pair. Then, LOOSE builds n-tuples of buckets (Lines 4-5). For each $\omega = (b_{1,l_1,l'_1}, \ldots, b_{n,l_n,l'_n})$, we obtain score bounds using bounds computed for each pair $(b_{i,l_i,l'_i}, b_{j,l_j,l'_j})$.

Algorithm 2 Strategies LOOSE, TWO-PHASE

Input: Boolean onePhase, Buckets $\{b_{i,l_i,l'_i} : i \in 1 \dots n\}$
Output: $\Omega_{k,\mathcal{S}}$

1: $L_2 \leftarrow$ all bucket pairs (b_i, b_j) s.t. $(i,j) \in E$
2: **for all** ω in L_2
3: Compute $\omega.UB$, $\omega.LB$ using solver
4: **for** ω in Ω
5: Compute $\omega.UB$, $\omega.LB$ using score bounds from $(b_i, b_j) \in L_2$
6: $L_m \leftarrow$TopBuckets(Ω)
7: **if** onePhase **then return** L_m
8: **for** $\omega \in L_m$
9: Compute $\omega.UB$, $\omega.LB$ using solver
return TopBuckets(L_m)

To calculate correct bounds, we rely on the monotonicity of $\mathcal{S}$. Without loss of generality, suppose that $\mathcal{S}$ is monotonically increasing. In the expression of $\mathcal{S}$, we replace each partial score $s\text{-}p_{(i,j)}(x_i, x_j)$ with the upper bound of the corresponding pair of bucket. Therefore $\mathcal{S}_{(i,j)\in E}((b_{i,l_i,l'_i}, b_{j,l_j,l'_j}).UB)$ is a correct upper-bound for ω. Then, LOOSE runs *getTopBuckets* on the generated bucket combinations (Line 6) and returns the selected combinations. The rationale behind LOOSE is that processing time can decrease significantly because (i) fewer bucket combinations need to be assigned a score bound (their number is $\mathcal{O}(|E| \cdot g^4)$) and (ii) the solver needs to assign only 4 variables when computing score bounds. A drawback of LOOSE is that the aggregation of bounds using $\mathcal{S}$ may result in loose bounds. We illustrate that in the following example.

Example. Figure 6 depicts a dataset with three buckets b_1, b_2, b_3 from C_1, C_2, C_3. We have $b_1 = (g_1, g_2)$, $b_2 = (g_2, g_3)$, $b_3 = (g_3, g_3)$, where $g_1 = [10, 20], g_2 = [20, 30], g_3 = [30, 40]$. Our query features scored predicates $s\text{-}starts_{(1,2)}(x, y)$ and $s\text{-}starts_{(2,3)}(y, z)$ and our aggregation function is the normalized sum. We use the scoring parameters $\{(\lambda_{equals}, \rho_{equals}), (\lambda_{greater}, \rho_{greater}) = \{(1,3), (0,4)\}$. LOOSE first computes bounds for $\omega_1 = (b_1, b_2)$. We have $\omega_1.UB = 1$ (because $s\text{-}starts_{(1,2)}(x_1, y_1) = 1$), and $\omega_1.LB = 0$ ($s\text{-}starts_{(1,2)}(x_1, y_2) = 0$). Then, LOOSE computes bounds for $\omega_1 = (b_2, b_3)$. We have $\omega_2.UB = 1$ ($s\text{-}starts_{(2,3)}(y_2, z_1) = 1$), and $\omega_2.LB = 0$ ($s\text{-}starts_{(2,3)}(y_2, z_2) = 0$). Then, LOOSE merges combinations ω_1, ω_2 in $\omega_3 = (b_1, b_2, b_3)$, and computes $\omega_3.UB = \mathcal{S}(1,1) = 1$, $\omega_3.LB = \mathcal{S}(0,0) = 0$. Yet, BRUTE-FORCE, computes $\omega_3.UB = 0.5$ because there is no (x, y, z) such that $s\text{-}starts_{(1,2)}(x, y) = s\text{-}starts_{(2,3)}(y, z) = 1$ given the buckets' bounds: it is impossible to have both $equals(\underline{x}, \underline{y}) = 1$ and $equals(\underline{y}, \underline{z}) = 1$ with $\underline{x} \in g_1, \underline{y} \in g_2$ and $\underline{z} \in g_3$. Thus, in this example, LOOSE returns an exact lower-bound and a loose upper-bound, while BRUTE-FORCE returns tight bounds.

These observations lead to propose a third strategy TWO-PHASE, that combines BRUTE-FORCE and LOOSE. TWO-PHASE is executed by Algorithm 2 when the flag onePhase is set to false. First, TWO-PHASE computes loose bounds to eliminate some bucket combinations (Lines 1-7), identically to LOOSE. Then, TWO-PHASE refines the bounds of the remaining combinations (Lines 8-9) to obtain exact bounds. The rationale behind TWO-PHASE is that its first phase may help reduce the number of bucket combinations that need to be assigned a score in the second phase, thus improving the solver's running time. Unlike LOOSE, TWO-PHASE returns tight bounds thanks to the second phase of the solver.

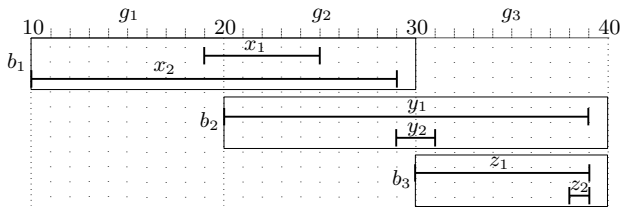


Figure 6: Example of Bucket Combinations

When selecting bucket combinations, *TKIJ* runs *TopBuckets* using one of the three strategies presented. Each strategy (i) employs the solver and (ii) executes *getTopBuckets* once or twice using loose or tight score bounds on bucket combinations.

3.4 Distributed Top-k Join Processing

The *TopBuckets* process generates $\Omega_{k,\mathcal{S}}$, a set of bucket combinations that are sufficient to accurately compute the top-k results. We now describe how *TKIJ* computes the top-k results (Steps (c)-(d)-(e) in Figure 5). We implement *TKIJ* on Map-Reduce [6]. Given a set of r reducers, *TKIJ* assigns each bucket combination $\omega \in \Omega_{k,\mathcal{S}}$ to a single reducer $r_j, j \in 1 \dots r$, that processes results in ω. The main challenge in distributed join processing is to devise an efficient workload assignment function. When performing large-scale joins, I/O often constitutes a major bottleneck. We first review existing assignment algorithms, then we consider the specifics of distributed top-k computation and show that it is essential to take scores into account when dividing the workload. We present *DistributeTopBuckets*, a novel function that focuses on assigning high-scoring results to each reducer, while minimizing I/O cost as a secondary objective. Finally, we present how an RTJ query is processed using appropriate Map-Reduce algorithms.

Existing I/O optimizations

When different reducers require the same chunk of data, this data is replicated in the shuffle phase of Map-Reduce, which increases input cost. Several distributed join algorithms, such as *RCCIS* [5] and the work of Afrati et al. [1] specifically aim at reducing that cost. In *TKIJ*, this corresponds to different reducers being assigned bucket combinations involving the same bucket. Other approaches focus on assigning a balanced load to each reducer [5, 24]. This ensures that the number of results generated by each reducer is comparable, so that no reducer will have a larger workload in output dominated tasks. Finally, some algorithms optimize both input and output costs simultaneously [30]. All these approaches are not directly applicable to our settings. They achieve optimizations for specific queries (equi-join [1], 2-way θ-join [24], m-way θ-join [30]). One close related work to ours [5] reduces I/O cost by leveraging the Boolean interpretation of Allen predicates. That is not directly applicable to scored predicates.

Top-k optimizations

TKIJ significantly differs from standard Map-Reduce-based join processes due to its ranked semantics. In *TKIJ*, each reducer processes a full RTJ query locally using the bucket combinations it receives (Figure 5d). Hence, it is important

Algorithm 3 DistributeTopBuckets (DTB)

Input: $\Omega_{k,\mathcal{S}}$
Output: M: assignments (bucket, reducer)
1: Sort $\Omega_{k,\mathcal{S}}$ by descending order of $\omega.UB$
2: $avgRes = \frac{\sum_{\omega \in \Omega_{k,\mathcal{S}}} \omega.nbRes}{r}$
3: **for all** $\omega \in \Omega_{k,\mathcal{S}}$
4: $r_j = getReducer(avgRes, \omega)$
5: **for all** bucket $b \in \omega$ ▷ Assign buckets in ω to r_j
6: $M \leftarrow M \cup (b, r_j)$
return M

to ensure that each reducer quickly identifies high-scoring results as it is usually the case in top-k processing [14, 15, 16, 26]. Therefore, the assignment of bucket combinations to reducers favors an even distribution of high-scoring results.

DTB algorithm

TKIJ relies on the *DistributeTopBuckets* algorithm (Algorithm 3) to assign bucket combinations from $\Omega_{k,\mathcal{S}}$ to reducers. Following the principles described above, *DTB* increases the probability that each reducer receives a fair share of high-scoring results. This step relies on the knowledge, for each bucket combination, of the number of results generated, as well as their score bounds. *DTB* first sorts $\Omega_{k,\mathcal{S}}$ by descending order of score upper-bound (Line 1) to access them according to their likelihood of generating high-scoring results. It then assigns each bucket combination to a reducer using the *getReducer* function (Algorithm 4), which returns a reducer among the ones that were assigned the fewest bucket combinations so far.
Furthermore, *DTB* opportunistically optimizes I/O cost. First, in the worst case, a reducer evaluates all the results it is assigned. If Ω_{r_j} is the set of bucket combinations assigned to a reducer r_j, then $\sum_{\omega \in |\Omega_{r_j}|} \omega.nbRes$ would be computed. *DTB* first computes the average number of results assigned to reducers (Algorithm 3, Line 2). Then *getReducer* ensures that reducers that are already assigned more than twice the average number of results are discarded (Algorithm 4, Lines 3, 7). This heuristic limits imbalance in a worst-case scenario. In the case, where several reducers have received the same number of bucket combinations, *getReducer* selects the reducer that was already assigned the largest fraction of current ω from previous overlapping bucket combinations (Algorithm 4, Lines 8-10). For a given w, *getReducer* evaluates for each reducer r_j the input cost of assigning w to r_j using $inCost(r_j, \omega)$. We define $inCost(r_j, \omega) = \sum_{b_{i,l,l'} \in \omega} |b_{i,l,l'}| \cdot \Phi(r_j, b_{i,l,l'})$ where $\Phi(r_j, b_{i,l,l'})$ returns 1 if $b_{i,l,l'}$ was already assigned to r_j, else 0. This process favors assignments that reduce replication cost. Having selected the reducer r_j that has to be assigned the current bucket combination ω, *DTB* stores all the assignments $(b_{i,l,l'}, r_j)$ where $b_{i,l,l'} \in \omega$ (Algorithm 3, Lines 5-6). These assignments determine to which reducers buckets are communicated, ensuring both output correctness and join processing efficiency.

Join Processing

The final phase of *TKIJ* first runs *DTB* using $\Omega_{k,\mathcal{S}}$ to determine data distribution among reducers. Then, a Map-Reduce phase processes the RTJ query locally. For each

Algorithm 4 getReducer

Input: ω bucket combination to assign, *avgRes* average number of results per reducer
Output: reducer to which ω has to be assigned
1: $min\_\omega\_assigned = +\infty$
2: **for** $j = 1$ to r ▷ Retrieve the minimum amount of bucket combinations assigned
3: **if** $r_j.nbRes < 2 \times avgRes$
4: $min\_\omega\_assigned = \min\{|\Omega_{r_j}|, min\_\omega\_assigned\}$
5: $minCost = +\infty$
6: **for** $j = 1$ to r ▷ Find the best reducer wrt $inCost(\omega, r_j)$
7: **if** $r_j.nbRes < 2 \times avgRes \wedge |\Omega_{r_j}| = min\_\omega\_assigned$
8: **if** $inCost(r_j, \omega) < minCost$
9: bestReducer=r_j
10: $minCost = inCost(r_j, \omega)$
return bestReducer

input interval x, a mapper computes the bucket b_{i_x,l_x,l'_x} in which x falls. Then, x is communicated to all reducers r_j that received b_{i_x,l_x,l'_x}. That way, each reducer r_j receives its share of input data, and a list of bucket combinations $\Omega_{r_j} \subseteq \Omega_{k,\mathcal{S}}$ whose results are potential top-k candidates. Once each reducer has processed locally the RTJ query, we run an additional Map-Reduce phase (Step (e) in Figure 5), that merges local results and returns the final top-k answers.

4. EXPERIMENTS

Platform. We conduct experiments on an 8-node industrial cluster with 6 workers. Each worker has 1 Intel Xeon E5-2650L (8 cores), 32GB RAM, 5TB disk. Machines run Centos 6.6 with Cloudera 5.2.5 and Hadoop 2.5.0. All presented results are averages of 5 consecutive runs.

Statistics collection. We use the same number of granules g for each collection. We observed that only the number of interval $|C_i|$ per collection had a significant impact on statistics collection time. Statistics collection lasted between 28s for $|C_i| = 2 \times 10^5$ and 36s for $|C_i| = 5 \times 10^6$. Since this task is only executed once as a pre-processing for a given dataset, we do not include it in query evaluation time.

Selection of bucket combinations. We implement a distributed and multi-threaded version of *TopBuckets*. We split the set of buckets $\mathcal{B}_1$ into 6 equal-sized groups $\mathcal{B}_{1,j}, j \in 1 \ldots 6$. Then each worker $j \in 1 \ldots 6$ runs a local version of *TopBuckets* using buckets in $\mathcal{B}_{1,j}$ and all buckets in $\mathcal{B}_i, i \in 2 \ldots n$. Thus, each worker has as input a disjoint set of possible bucket combinations. We use Choco [25], a software for constraint programming to compute score bounds for each bucket combination. Each execution of the solver is handled in a separate thread. A second phase of *TopBuckets* merges local *TopBuckets* results on a selected worker and returns the set of best bucket combinations $\Omega_{k,S}$.

Distributed join processing. We run *TKIJ* using 24 reducers. Local query execution accesses the received bucket combinations by descending order of score upper-bounds. It then uses R-Trees to access intervals in memory. For an interval x_i and a score value v, it queries the R-Tree and returns only intervals x_j s.t. $s\text{-}p_{(i,j)}(x_i, x_j) \geq v$.

Id	Scored Temporal Predicates In $\mathcal{Q}$. $x_i \in C_i \ \forall i \in 1 \ldots n$.
$\mathcal{Q}_{b,b}$	$s\text{-}before(x_1, x_2), s\text{-}before(x_2, x_3)$
$\mathcal{Q}_{f,f}$	$s\text{-}finishedBy(x_1, x_2), s\text{-}finishedBy(x_2, x_3)$
$\mathcal{Q}_{o,o}$	$s\text{-}overlaps(x_1, x_2), s\text{-}overlaps(x_2, x_3)$
$\mathcal{Q}_{s,f,m}$	$s\text{-}starts(x_1, x_2), s\text{-}finishedBy(x_2, x_3), s\text{-}meets(x_1, x_3)$
$\mathcal{Q}_{s,s}$	$s\text{-}starts(x_1, x_2), s\text{-}starts(x_2, x_3)$
$\mathcal{Q}_{b*}$	$s\text{-}before(x_1, x_2), \ldots, s\text{-}before(x_1, x_n)$
$\mathcal{Q}_{o*}$	$s\text{-}overlaps(x_1, x_2), \ldots, s\text{-}overlaps(x_1, x_n)$
$\mathcal{Q}_{m*}$	$s\text{-}meets(x_1, x_2), \ldots, s\text{-}meets(x_1, x_n)$
$\mathcal{Q}_{f,b}$	$s\text{-}finishedBy(x_1, x_2),\ s\text{-}before(x_2, x_3)$
$\mathcal{Q}_{o,m}$	$s\text{-}overlaps(x_1, x_2),\ s\text{-}meets(x_2, x_3)$
$\mathcal{Q}_{s,m}$	$s\text{-}starts(x_1, x_2),\ s\text{-}meets(x_2, x_3)$
$\mathcal{Q}_{jB,jB}$	$s\text{-}justBefore(x_1, x_2),\ s\text{-}justBefore(x_2, x_3)$
$\mathcal{Q}_{sM,sM}$	$s\text{-}shiftMeets(x_1, x_2),\ s\text{-}shiftMeets(x_2, x_3)$

Table 1: Queries

Id	$(\lambda_{equals}, \rho_{equals})$	$(\lambda_{greater}, \rho_{greater})$
$\mathcal{P}_1$	(4,16)	(0,10)
$\mathcal{P}_2$	(0,16)	(2,8)
$\mathcal{P}_3$	(4,12)	(0,8)
$\mathcal{P}_B$	(0,0)	(0,0)

Table 2: Scored Predicates Parameters

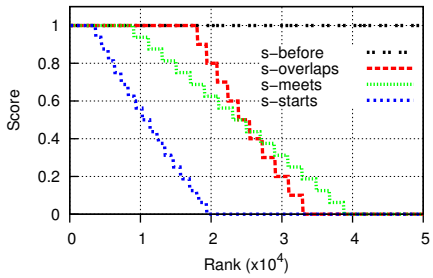
Queries. Tables 1 and 2 summarize queries and score parameters. We use $\mathcal{S} = \frac{\sum_{(i,j)\in E} s\text{-}p_{(i,j)}(x_i,x_j)}{|E|}$ to compute the score of a query result $(x_1, \ldots, x_n)$. Unless otherwise specified, $k = 100$.

4.1 Summary of Results

We show that *TKIJ* processes various RTJ queries efficiently on both synthetic data and real network traffic logs. *TKIJ* scales to collections of up to 5 million intervals ($|C_i| \in [1M, 5M]$) and efficiently returns the top-k results for $k \in [10, 10^5]$. We show that our workload distribution approach, *DistributeTopBuckets*, that fairly distributes high-scoring results, outperforms a more naive approach based on the *LPT* algorithm. That is particularly useful for queries that return few high-scoring results, such as those featuring equality-based predicates (e.g *starts*). We observe that the efficiency of *DistributeTopBuckets* decreases with coarser statistics. We also show that as the number of collections or the number of predicates in a query vary, *TopBuckets* efficiently prunes buckets combinations. Experiments on network traffic data show that on queries featuring *before* or *overlaps*, *TopBuckets* can select few bucket combinations guaranteeing *TKIJ* to return high-scoring results. We observe that a higher number of granules g (finer statistics) helps prune unnecessary results, which improves overall join processing time. However, it also makes pruning computation with *TopBuckets* slower. Hence, we run experiments to find a sweet-spot value for g.

4.2 Synthetic Data

To generate synthetic data, we use the same parameters as in previous work [5]. We use a pseudo-random uniform generator to get intervals' startpoints and lengths in specified ranges (respectively $s = [0, 10^5]$ and $w = [1, 100]$). Intervals' endpoints are integers. We vary the number $|C_i|$ of intervals per collection, and the number n of collections. A collection of 5M intervals measures ≈113MB (text format).



Parameters: $|C_i| = 10^4, \mathcal{P} = \mathcal{P}_1$

Figure 7: Synthetic Data - Score Distribution

4.2.1 Score Distribution

We conduct preliminary experiments to get an insight of the score distribution of a set of results using scored temporal predicates. Especially, we want to measure the share of results that have a high score. We want to verify that the fewer the number of high-scoring results, the less likely a worker will receive high scoring results from *DTB*. We compute all the combinations $(x_1, x_2) \in C_1 \times C_2$ and we evaluate each result with scored predicates $s\text{-}before(x_1, x_2)$, $s\text{-}meets(x_1, x_2)$, $s\text{-}overlaps(x_1, x_2)$ and $s\text{-}starts(x_1, x_2)$.

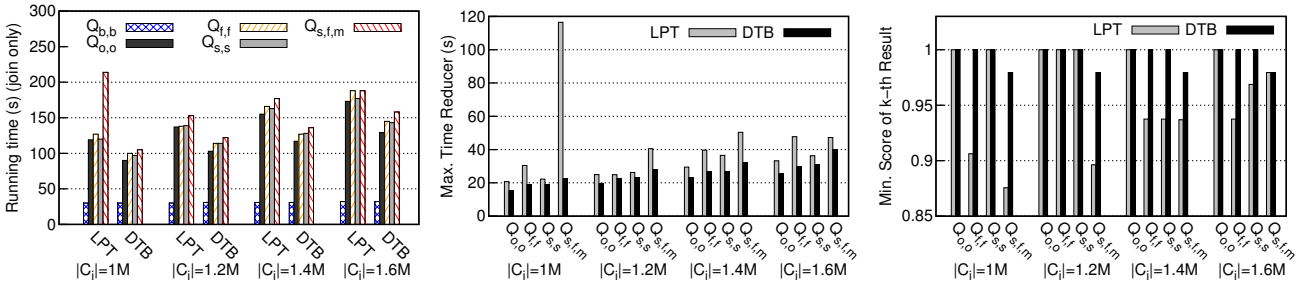
On Figure 7, we plot scores for the top-50000 results. *s-before* is the predicate with the largest number of high-scoring results (scores of top-50000 results equal 1.0), since a single inequality on endpoints is required. More high-scoring results can be found with *s-overlaps* (≈ 18,000 results), that evaluates only inequality on endpoints, than with *s-meets* (≈9000), where an equality is required. Fewer results are assigned a high score when *s-starts* is used since it requires both equality and inequality on endpoints. Thus, we can expect a faster join processing on queries using only inequality-based predicates where more high-scoring results can be found, since high-scoring results favor early termination of local top-k processing.

4.2.2 Workload Distribution

We conduct experiments to validate our workload distribution approach. We analyze executions of *TKIJ* using *DTB*, our workload distribution algorithm, and using a more straightforward algorithm.

LPT. In the context of task scheduling, the *LPT* (*Longest Processing Time*) heuristic aims to minimize scheduling time on parallel machines [11]. *LPT* executes tasks in descending order of processing time. In our context, a naive approach would minimize the maximum number of candidate join results that a reducer has to process, so as to reduce the duration of the longest task. With *LPT*, bucket combinations are analogous to tasks that we want to assign to a set of reducers. We sort the set of bucket combinations by descending order of number of results ($\omega.nbRes$) and assign each one to the least loaded reducer.

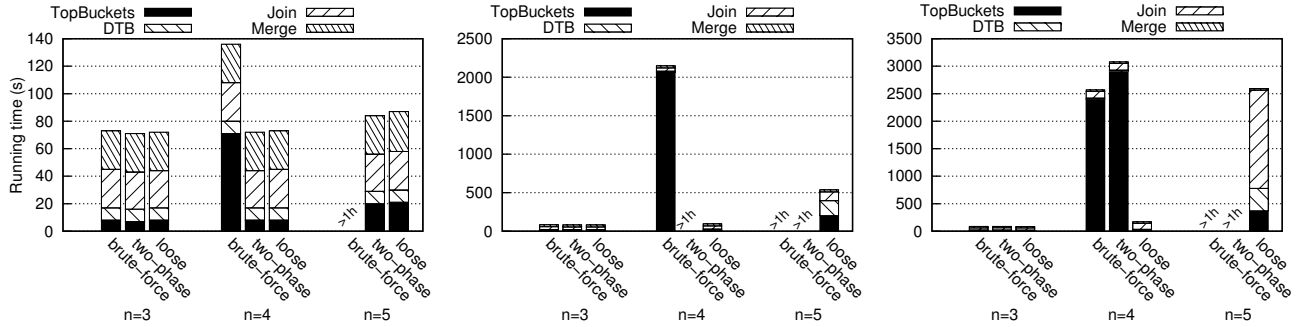
Results. Figure 8a presents the running time of the join phase on all queries, where $|C_i|$ varies from 1×10^6 to 1.6×10^6. On $\mathcal{Q}_{b,b}$, running time is identical for *LPT* and *DTB*, since a single bucket combination is selected and a large number of results with maximum score can be quickly found during the join phase. On other queries, *DTB* outperforms



(a) Running Time (b) Max Running Time of Reducers (c) Minimum Score of k-th Result

Parameters: $g = 20, k = 1000, \mathcal{P} = \mathcal{P}_2$, *TopBuckets* : LOOSE

Figure 8: Synthetic Data - Workload Distribution



(a) $\mathcal{Q}_{b*}$ (b) $\mathcal{Q}_{o*}$ (c) $\mathcal{Q}_{m*}$

Parameters: $g = 15, k = 100, |C_i| = 2 \times 10^5, \mathcal{P} = \mathcal{P}_1$

Figure 9: Synthetic Data - Detailed execution time, all *TopBuckets* strategies

LPT for two reasons. Firstly, *LPT* incurs a higher shuffle cost (on average 43% higher). When assigning a bucket combination to a reducer, *DTB* favors assignments that lessen shuffle cost. *LPT* favors the assignment of bucket combinations with a large number of results to the least loaded reducers. Hence, buckets have a higher probability to be sent to several reducers with *LPT* than with *DTB*. Secondly, *LPT* does not necessarily give a fair share of high-scoring results to each reducer. Figure 8b shows the running time of the longest reducer task (we omit $\mathcal{Q}_{b,b}$ where *LPT* and *DTB* perform equally for the reason exposed above). *DTB* always outperforms *LPT* because it increases the probability that all reduce tasks terminate early since they can all find high-scoring results. This difference is exacerbated on query $\mathcal{Q}_{s,f,m}$ with $|C_i|=1M$. Here, the few results that satisfy best all 3 predicates featured in $\mathcal{Q}_{s,f,m}$ are better distributed using *DTB*. On Figure 8c, we represent the minimum score of the k^{th} result among the results returned by reducers. These results support our observation: the score of returned results is higher when distribution is defined using *DTB*, while unnecessary results with lower scores are returned with *LPT*.

4.2.3 TopBuckets Strategies

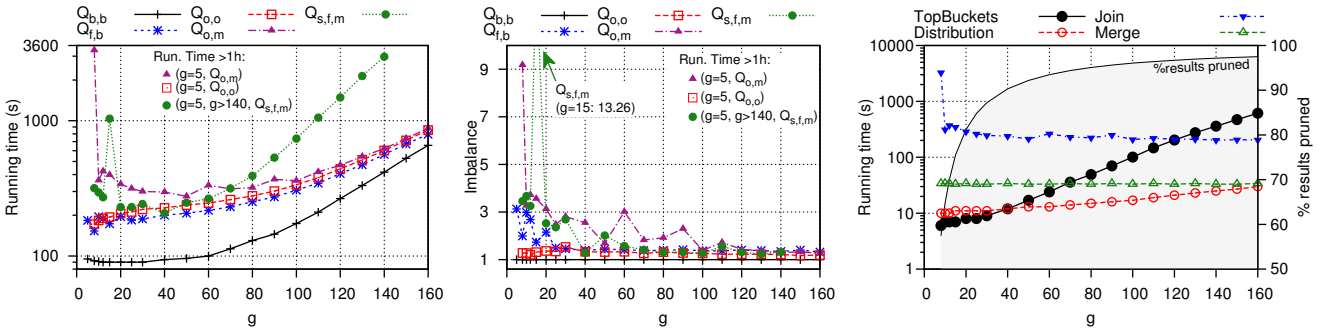
We conduct experiments on the *TopBuckets* strategies exposed in Section 3.3. We vary the number of collections n using queries $\mathcal{Q}_{b*}$, $\mathcal{Q}_{o*}$ and $\mathcal{Q}_{m*}$.

Figure 9 summarizes the results. We do not report results where running time exceeds 1 hour. Experiments show the inefficiency of BRUTE-FORCE and TWO-PHASE. On these strategies, n-tuples of buckets, where $n \in 3 \dots 5$, are assigned a score bound by the solver. With BRUTE-FORCE, the running time of *TopBuckets* quickly increases (solid black box on Figure 9) with n, because the solver needs to compute score bounds for a large number of bucket combinations, each one requiring to assign $2n$ variables. The TWO-PHASE strategy only beats BRUTE-FORCE on $\mathcal{Q}_{b*}$ (Figure 9a) where its first phase prunes a large share (more than 99% for any n) of possible bucket combinations, thus limiting the running time of the second phase, that computes exact bounds using a smaller set of bucket combinations. On others queries, TWO-PHASE does not improve running time: the first phase does not prune enough combinations (e.g. 52% on $\mathcal{Q}_{m*}$ for $n = 4$) to lessen the cost of the second phase where remaining combinations need to be assigned tight score bounds. The LOOSE strategy is the most efficient: (i) loose bounds do not impact significantly join processing time as a large share of potential results (e.g. 81% on $\mathcal{Q}_{o*}$ for $n = 4$) remained pruned and (ii) *TopBuckets* scales with the number of collections n. In the remainder of our experiments, we use LOOSE as the *TopBuckets* strategy.

4.2.4 Number of Granules

Since the second phase of *TKIJ* relies on collected statistics to prune the input space and distribute the workload, we expect *TKIJ* to depend on the granularity of statistics (e.g. coarse or fine-grained). We conduct experiments to validate this intuition, varying the number of granules g.

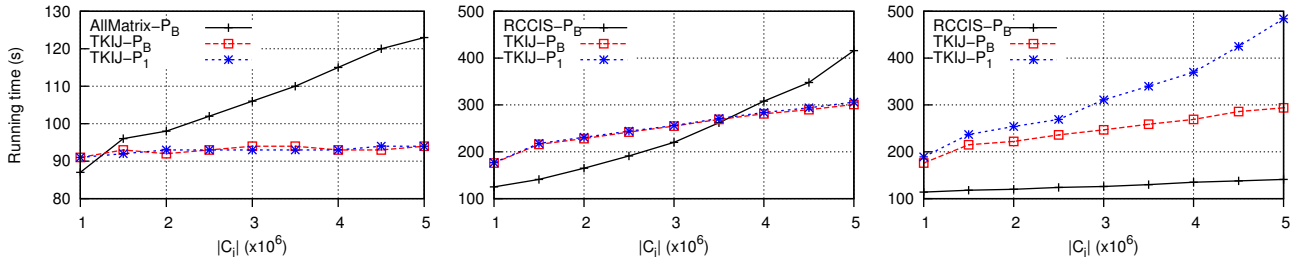
We present respectively on Figures 10a, 10b and 10c the



(a) Running Time (b) Imbalance (c) Detailed Running Time for $\mathcal{Q}_{o,m}$

Parameters: $k = 100, |C_i| = 2 \times 10^6$, $\mathcal{P} = \mathcal{P}_1$, *TopBuckets*: LOOSE

Figure 10: Synthetic Data - Effect of number of granules g



(a) $\mathcal{Q}_{b,b}$ (b) $\mathcal{Q}_{o,o}$ (c) $\mathcal{Q}_{s,m}$

Parameters: $g = 40$, $k = 100$, *TopBuckets*: LOOSE

Figure 11: Synthetic Data - Scalability

total running time of a range of queries, the load imbalance of the join phase computed using $\frac{Max\ Time\ Reducer}{Average\ Time\ Reducer}$ and the detailed running time for query $\mathcal{Q}_{o,m}$. We do not report results for executions where the total running time exceeds 1 hour. On queries that return the fewest high-scoring results ($\mathcal{Q}_{o,m}$, $\mathcal{Q}_{s,f,m}$), we observe on Figure 10a that with a lower g (coarse statistics), running time degrades. *TKIJ* suffers here from poor workload distribution. This is expected as with fewer granules, we have fewer bucket combinations and especially fewer high-scoring ones. As *TKIJ* relies on a round-robin distribution of high-scoring buckets combinations, there is a lower probability to provide each reducer high-scoring results. Hence, we can observe on Figure 10b the imbalance that is more variable when g decreases. As illustrated on Figure 10c, when g increases, the local join processing is faster, thanks to a better workload distribution and to a larger pruning of unnecessary results. 81% of potential results are pruned for $g = 20$, while it is 96% for $g = 100$ (grey filled curve on Figure 10c). Yet, the pruning process *TopBuckets* is slower when g increases and thus worsen the overall response time. Note that because it features more predicates, query $\mathcal{Q}_{s,f,m}$ requires to evaluate more bucket pairs during the *TopBuckets* process. Thus, *TopBuckets* running time increases faster with g than on others queries, hence the impact on the overall response time.

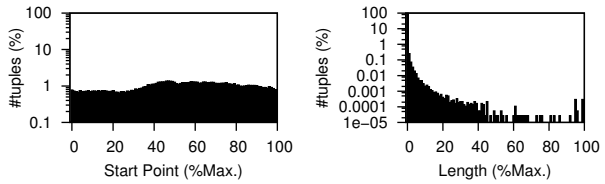
For queries $\mathcal{Q}_{b,b}$ and $\mathcal{Q}_{o,o}$, coarse statistics have nearly no effect since a large number of high-scoring results can be found during the join phase (except when $g = 5$ where workload distribution fails for query $\mathcal{Q}_{o,o}$: a reducer does not quickly find high-scoring results). Finally, we observe that a number of granules $g \approx 40$ provides the best trade-off for the various queries that we experimented. Future investigations include the design of benchmark approaches or the adaptation of optimization techniques [7] to compute the optimal number of granules that minimizes execution time of *TKIJ*.

4.2.5 *Scalability*

We vary the size of collections $|C_i|$ to evaluate the scalability of *TKIJ*. We compare *TKIJ* to state-of-the-art competitors.

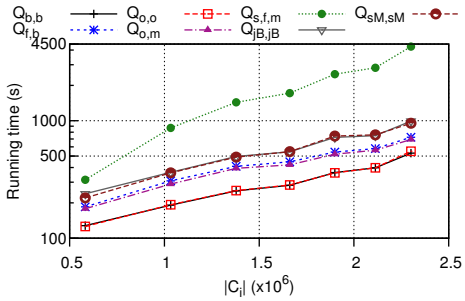
As a baseline, we borrow algorithms from related work on processing interval joins on Map-Reduce [5]. In this work, algorithms *RCCIS* and *All-Matrix* are designed to process interval joins using Allen predicates. In our settings, we use these algorithms to return only results that satisfy all the Boolean predicates of a RTJ query (i.e. a subset of top-k results). We also impose reducers to stop join processing if k results are found. Then, we merge and sort local results using a final Map-Reduce phase, identically to *TKIJ*.

RCCIS handles only *colocation* predicates where intervals intersect (e.g. *overlaps, meets*). *All-Matrix* handles only *sequence* predicates (*before, after*). *RCCIS* and *All-Matrix* also partition the temporal range using contiguous granules. For *RCCIS*, we set the number of granules to 24 which implies that 24 reducers are used. For *All-Matrix*, the number of reducers depends on the number of granules and of collections. We used 4 granules with $n = 3$, yielding 20 reducers. For *TKIJ*, we conduct a first set of experiments with



(a) Start Point (b) Length

Figure 12: Network Traffic Data Distribution



Parameters: $g = 40, k = 100, \mathcal{P} = \mathcal{P}_3$, *TopBuckets*: LOOSE

Figure 13: Network Traffic Data - Scalability

score parameters $\mathcal{P}_B = \{(0,0),(0,0)\}$ (see Table 2). We are hence using a Boolean interpretation of predicates. Because *TKIJ* must return k results, if only $k' < k$ results satisfy the Boolean predicates (with $\mathcal{S}(t) = 1.0$), $k - k'$ other results that do not satisfy at least one predicate will be returned (with $\mathcal{S}(t) < 1.0$). Thus, *TKIJ* may need to return more results than *All-Matrix* or *RCCIS*, that only return results fully satisfying all Boolean predicates. Hence, results of each algorithm are not directly comparable. For *TKIJ*, we conduct a second set of experiments using scored predicates with the score parameters $\mathcal{P}_1$.

Results. We present total running times on Figure 11. For query $\mathcal{Q}_{b,b}$ (Figure 11a), *TKIJ* remains nearly constant since *TopBuckets* returns only 1 bucket combination. Thus, *TKIJ* processes only a small share of input data. *All-Matrix* shuffles intervals belonging to all possible results, hence running time increases with $|C_i|$. For query $\mathcal{Q}_{o,o}$ (Figure 11b), *TopBuckets* selects more combinations to process. As the number of intervals per bucket increases with $|C_i|$, more data is shuffled and processed during the local join processing, hence the running time increases linearly with $|C_i|$. Figure 11b also shows that *TKIJ* outperforms *RCCIS* on $|C_i| > 3.5 \times 10^6$. In *RCCIS*, a first Map-Reduce phase builds intermediate results to determine which intervals need to be replicated to ensure output correctness in the join phase. Thus, its running time increases with $|C_i|$. Meanwhile, *TKIJ* decides which tuples should be combined on the basis of *TopBuckets*, which does not depend on $|C_i|$ and is on average 93% faster than the first phase of *RCCIS*. On $\mathcal{Q}_{s,m}$ (Figure 11c), we do not observe this phenomenon anymore: *RCCIS* first phase is faster (there are fewer intermediate results), while *TKIJ*'s join phase is longer (there are fewer high-scoring results). We observe a notable difference on query $\mathcal{Q}_{s,m}$ with scored and with Boolean predicates. The local join processing explains this difference. In the Boolean case, *TKIJ* focuses on building results where join conditions are satisfied (whose score is strictly positive), thus limiting the search space. With the approximate interpretation of predicates, more combinations need to be considered because tolerance on endpoints incurs a higher number of results with a strictly positive score. Thus, more intermediate results are computed. On query $\mathcal{Q}_{o,o}$, a large number of results have the highest score (Figure 7) which incurs major pruning. That justifies the absence of a significant difference between the scored and Boolean cases.

4.2.6 *Effect of k*

We conducted experiments where k varies in $[10, 10^5]$ on a range of queries ($\mathcal{Q}_{b,b}$, $\mathcal{Q}_{o,o}$, $\mathcal{Q}_{s,f,m}$, $\mathcal{Q}_{f,b}$, $\mathcal{Q}_{o,m}$) with $|C_i| = 2 \times 10^6$. We observed that *TKIJ* is almost constant on all queries and all values of k. Actually, a large number ($> 10^{13}$) of potential results fall in each bucket combination. Thus, the set of selected bucket combinations remains the same for $k \in [10, 10^5]$ since we can always guarantee to return the correct top-k results.

4.3 Network Traffic Data

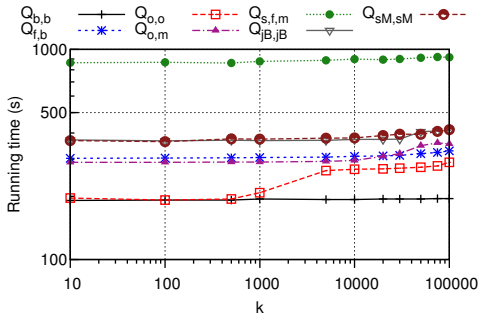
4.3.1 *Data*

We use network traffic data collected on firewall logs of a data hosting company. Each log contains packets exchanged between servers and clients ($\approx$ 5GB, 100M packets per day). Each packet has a timestamp (seconds). We selected one log and built a list of connections by grouping packets exchanged between a pair (server, client). Only consecutive packets whose timestamps are within a time interval $[0, 60]$ are grouped. A connection [*client*, *server*, *start*, *end*] represents the activity of *client* on *server*, where the first packet was sent or received at timestamp *start* and the last one at timestamp *end*. The dataset obtained includes 3,636,814 intervals ($\approx$ 83MB), whose minimum, maximum, and average length are respectively 1, 86,459 and 54 seconds. Figure 12 shows the distribution of start points and lengths. Then, we copy each list of connections 3 times and process 3-way queries. We are interested in real-life scenarios occurring in network traffic analysis, hence we process queries $\mathcal{Q}_{jB,jB}$ and $\mathcal{Q}_{sM,sM}$ (Table 1). Query $\mathcal{Q}_{jB,jB}$ returns the sequences of connections that closely follow each other, while $\mathcal{Q}_{sM,sM}$ returns sequences where a delay was observed between two connections.

4.3.2 *Scalability*

We verify that *TKIJ* scales with various dataset sizes. When generating connections, we use various randomly selected samples on the log file used. We pick from 5% to 35% of input data. We obtain collections of connections whose number of intervals varies from 0.58×10^6 to 2.31×10^6.

We present total running times on Figure 13. We note that running time increases faster than what was observed on synthetic data. Here, when we use larger samples on input data, we have more buckets containing at least an interval. When $|C_i| = 0.58 \times 10^6$, there are 151 buckets containing at least an interval, while there are 296 for $|C_i| = 2.31 \times 10^6$. Thus, *TopBuckets* has to process more bucket combinations with higher $|C_i|$. On query $\mathcal{Q}_{s,f,m}$, that features more predicates, the time taken by *TopBuckets* is dominant (e.g. 82% of overall response time for $|C_i| = 1.38 \times 10^6$). Hence, overall response time increases faster on query $\mathcal{Q}_{s,f,m}$ than on all



Param.: $|C_i| = 1.03 \times 10^6$, $g = 40$, $\mathcal{P} = \mathcal{P}_3$, *TopBuckets*: LOOSE

Figure 14: Network Traffic Data - Effect of k

other queries. We also observe that while $\mathcal{Q}_{o,o}$ lasts longer on synthetic data, *TKIJ* performs similarly on $\mathcal{Q}_{b,b}$ and $\mathcal{Q}_{o,o}$ on real data. That can be explained by the fact that the real dataset contains long intervals (Figure 12). These intervals fall into buckets built with granules that are far apart (e.g. $b_1 = ([2160, 4320], [19440, 21600])$ or $b_2 = ([8640, 10800], [38880, 41040])$). Thus, we can find bucket combinations (e.g. $\omega = (b_1, b_2)$) whose results (x, y) are guaranteed to have a high score *s-overlaps*(x, y). Then, *TopBuckets* returns less bucket combinations, reducing the search space while guaranteeing correctness.

4.3.3 *Effect of k*

We verify *TKIJ* on various values of k. We present running times on Figure 14. For all queries except $\mathcal{Q}_{o,o}$, we observe that *TKIJ* remains nearly constant when $k \leq 5000$. Then, the running time increases slowly when $k > 5000$: as more results need to be returned, more intermediate results are built before termination especially with queries having fewer high-scoring results. On $\mathcal{Q}_{o,o}$ we observe that *TKIJ* increases slightly between $k = 1000$ and $k = 5000$. That is explained by an increase (from 643 to 41,272) in the number of bucket combinations $|\Omega_{k,\mathcal{S}}|$ necessary to return the correct top-k results, which in turn increases the number of intermediate results.

5. RELATED WORK

Three research areas relate to our work, however none of them addresses the RTJ problem.

Interval Joins. The closest work to ours [5] addresses the processing of multi-way joins on Map-Reduce for Allen Boolean predicates [2]. The first algorithm, *RCCIS*, solves *colocation* queries, where all predicates require intervals to have a non-empty intersection. *RCCIS* reduces the amount of data shuffled in Map-Reduce by sending to the same reducer intervals that are most likely to be colocated. The second algorithm, *All-Matrix* handles *sequence* queries. Because such queries imply unavoidable replication, *All-Matrix* focuses on load balancing. None of those algorithms is applicable to solving the RTJ problem as they do not handle scored results.

Although flexible interpretations of Allen predicates were proposed in approximate reasoning [12, 23, 27], none of them designed join algorithms. A series of work on spatio-temporal data focused on efficiently retrieving *overlapping* objects with Boolean semantics. Objects are stored in partitions [7, 21], or tree structures [13], such as the R-Tree [3, 19] or the quadtree [17]. A more recent investigation proposed a compound index structure using segment trees to find intervals that *intersect* a query-interval in a key-value cloud-store [28]. In summary, these studies focus on reducing I/O and support only Boolean overlaps and intersections.

Top-k Processing. Instance-optimal algorithms [14, 15] were proposed for *rank-joins* ranging from centralized implementations for *HRJN* [20] and *Pull-Bound Rank Join* [26] in [16], to distributed ones in [10, 22]. Specifically, in NoSQL databases, the *BFHM* algorithm [22] places each tuple in a bucket that depends on its score. Tuples are compressed using Bloom-filters allowing to select the *best* buckets first then retrieve tuples to be joined from the database. Because a Bloom-filter yields false positives, *BFHM* may require several iterations. Our work differs in two aspects. First, our scores are predicate-dependent and are not known *a priori*. Second, reiterating join processing in our case would incur too high an overhead.

Load Balancing. Load balancing is a common concern in parallel top-k processing. RanKloud [4] computes data statistics to retrieve an estimation of the *kth* join score. Then, only the part of input data whose score is above the estimated one is uniformly distributed and processed in parallel on a set of workers. Similarly, we rely on computing statistics to avoid processing useless data. However, while RanKloud outputs approximate results, our algorithm guarantees to return exact top-k results. Another work [8] proposes a partitioning scheme on Map-Reduce based on partitioning for parallel skyline query processing [29]. The *angle-based partitioning* distributes evenly the volume that contains points near the best possible point, increasing the probability that skyline points will fall evenly in each partition. This idea is reintroduced in the context of top-k joins [8] and is shown to be superior to a cardinality-based partitioning. Although we share the same intuition, we cannot directly apply this technique since it assumes that partial scores are known *a priori* while in our case they are predicate-dependent.

6. CONCLUSION

In this paper, we introduce RTJ queries using scoring functions reflecting the degree of satisfaction of a join predicate. We design *TKIJ*, a parallel multi-way top-k interval join algorithm on Map-Reduce. *TKIJ* relies on an offline collection of statistics to prune the search space and a workload distribution scheme appropriate to top-k processing. We conduct experiments on synthetic data that validate our approach and show the efficiency of *TKIJ* on various queries. We observe the same effectiveness of *TKIJ* on real network traffic logs. Future investigations include the integration of interval attributes (e.g. IP address for a connection) in the join conditions, to build hybrid queries, similarly to previous work [5]. We also plan to pursue investigations on various distributed platforms. Although robust and scalable, Map-Reduce suffer from limitations on complex queries [9]. For instance, a field of investigation is the design of algorithms that take advantage of communication between workers.

7. ACKNOWLEDGMENTS

This work was partially funded by the Datalyse PIA project.

8. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] K. S. Candan, J. W. Kim, P. Nagarkar, M. Nagendra, and R. Yu. Rankloud: Scalable multimedia data processing in server clusters. *IEEE MultiMedia*, 18(1):64–77, 2011.

[5] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing interval joins on map-reduce. In *EDBT*, pages 463–474, 2014.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *SIGMOD*, pages 1459–1470, 2014.

[8] C. Doulkeridis and K. Nørvåg. On saying "enough already!" in mapreduce. In *Cloud-I*, 2012.

[9] C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in mapreduce. *VLDB J.*, 23(3):355–380, 2014.

[10] C. Doulkeridis, A. Vlachou, K. Nørvåg, Y. Kotidis, and N. Polyzotis. Processing of rank joins in highly distributed systems. In *ICDE*, pages 606–617, 2012.

[11] M. Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009.

[12] D. Dubois, A. HadjAli, and H. Prade. Fuzziness and uncertainty in temporal reasoning. *J. UCS*, 9(9):1168, 2003.

[13] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *SIGMOD*, pages 683–694, 2004.

[14] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[16] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*, pages 415–428, 2009.

[17] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[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] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[20] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[21] H. Lu, B. C. Ooi, and K. Tan. On spatially partitioned temporal join. In *VLDB*, pages 546–557, 1994.

[22] N. Ntarmos, I. Patlakas, and P. Triantafillou. Rank join queries in nosql databases. *PVLDB*, 7(7):493–504, 2014.

[23] H. J. Ohlbach. Relations between fuzzy time intervals. In *TIME*, pages 44–51, 2004.

[24] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.

[25] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[26] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, pages 43–52, 2008.

[27] S. Schockaert, M. D. Cock, and E. E. Kerre. Fuzzifying allen's temporal interval relations. *IEEE T. Fuzzy Systems*, 16(2):517–533, 2008.

[28] G. Sfakianakis, I. Patlakas, N. Ntarmos, and P. Triantafillou. Interval indexing and querying on key-value cloud stores. In *ICDE*, pages 805–816, 2013.

[29] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, pages 227–238, 2008.

[30] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.