Information Systems 71 (2017) 164–181

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Efficiently answer top-k queries on typed intervals

Jianqiu Xu[a,*], Hua Lu[b]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
[b] Department of Computer Science, Aalborg University, Denmark

ARTICLE INFO

Article history:
Received 27 April 2016
Revised 7 April 2017
Accepted 15 August 2017
Available online 19 August 2017

ABSTRACT

Consider a database consisting of a set of tuples, each of which contains an interval, a type and a weight. These tuples are called *typed intervals* and used to support applications involving diverse intervals. In this paper, we study top-*k* queries on typed intervals. The query reports *k* intervals intersecting the query time, containing a particular type and having the largest weight. The query time can be a point or an interval. Further, we define top-*k* continuous queries that return qualified intervals at each time point during the query interval. To efficiently answer such queries, a key challenge is to build an index structure to manage typed intervals. Employing the standard interval tree, we build the structure in a compact way to reduce the I/O cost, and provide analytically derived partitioning methods to manage the data. Query algorithms are proposed to support point, interval and continuous queries. An auxiliary main-memory structure is developed to report continuous results. Using large real and synthetic datasets, extensive experiments are performed in a prototype database system to demonstrate the effectiveness, efficiency and scalability. The results show that our method significantly outperforms alternative methods in most settings.

1. Introduction

Intervals representing axis-parallel line segments have been widely used in a plethora of application domains. In temporal and multi-version databases, intervals are typically defined as transaction time and valid time ranges for recording changes (update, insertion or deletion), versions and the lifetime of objects [9,11,30,32,38]. In spatial and geographic information systems, intervals occur as line segments on a space-filling curve, e.g., modeling a printed circuit board [8,19]. Intervals also play a pivotal role in constraint databases [26].

In the literature, a number of operators have been studied on querying intervals such as intersecting [18], stabbing [5], splitters [30], and joins [13,17]. This work differs from them by investigating top-k queries on intervals which are associated with types and weights. Recent advances in sensing technologies have made collecting data with extensive information in ease. In real applications, intervals with diverse types may be collected due to different data sources. The system should be able to represent and manage the data for further queries. To the best of our knowledge, typed intervals have not been considered before.

In this paper, we investigate a database storing a set of tuples, each of which contains an interval consisting of start and end points, a type and a weight. Typed intervals enrich the data representation and support applications requiring different kinds of intervals, e.g., various genome intervals in genomics datasets, different versions of data items, and line segments categorized into several groups. Various choices are provided on the website "www.booking.com " for tourists such as five-star hotels, apartments and motels, and room rates change over time (e.g., hot and cold seasons, weekdays and weekends). The system needs to manage a large amount of typed intervals representing different hotel room rates. We study top-*k* queries on typed intervals in the paper. Formally, given a query time and a type, the system reports *k* intervals fulfilling the condition: (i) intersecting the query time; (ii) containing the type; and (iii) having the largest weight. To help understand the problem, we give some application examples.

Example 1. Fig. 1 shows a running example. The database stores a set of computer science projects. Each tuple keeps record of the lifetime, the project category and the budget (weight). There are totally four types of projects: {AI, DB, DM, OS}. A top-*k* query is *"return the top-1 DB project running at the time37"*, denoted by *Q*(37, DB, 1). Three intervals intersecting the time: {o_1, o_7, o_8}. However, o_1 is not reported because it is not a DB project. The system returns o_7 as the result because its weight is larger than o_8.

Example 2. In traffic monitoring systems, to analyze vehicles appearing in certain areas, we need to distinguish the different vehi-

* Corresponding author.
E-mail addresses: jianqiu@nuaa.edu.cn (J. Xu), luhua@cs.aau.dk (H. Lu).

http://dx.doi.org/10.1016/j.is.2017.08.005
0306-4379/

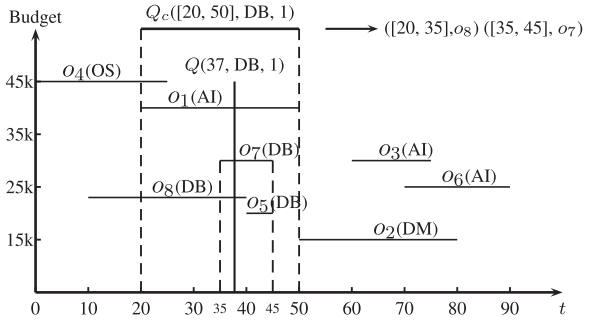


Fig. 1. A running example.

cles such as {*Taxi, Bus, Truck, Car*}. The database stores the number of vehicles appearing in a district. Such a value changes over time such that one district will have a sequence of typed intervals. Each tuple corresponds to a district and records the number of vehicles having the same type during a time interval. A top-k query is *"return the district with the largest number of trucks at 7:00am"*.

Such applications impose new challenges regarding indexing and querying intervals, in particular, supporting combined selections on different attributes. We define the query time by a point or an interval. The type predicate is included. Continuous queries are also studied to report top-k intervals at each time point during the query interval. This complicates the evaluation because the result changes at certain time points. Consider Q_c([20, 50], DB, 1) by referring to Fig. 1. The query aims to return the maximum budget DB project at each time during [20, 50]. One can see that o_7 is the DB interval with the maximum budget but is only valid during [35, 45]. The system will return o_8 during [20, 35].

To efficiently answer top-k queries, the key issue is to develop an index structure that can (i) efficiently index intervals for intersecting queries; (ii) well manage different types such that we can quickly find intervals with a particular type; and (iii) order intervals on weights to minimize the number of accessed intervals as only k intervals are reported. It is not difficult to achieve each of them individually, but a complex task to well support all of them. One can treat each condition as a predicate and we need an access structure that allows a combined evaluation of three predicates. This motivates us to develop an efficient index structure for typed intervals.

We choose Edelsbrunner's interval tree [14] as the basic structure. This structure and its variants have been commonly used in existing works [4,9,18,28], and the interval tree provides the primitive functionality needed in solving our problem. In principle, an interval tree is a binary tree that serves as the *primary* structure. Each node maintains a value called the *split* point and two lists of sorted intervals that intersect the split point, called the *secondary* structure. Intervals smaller and larger than the split point are stored in the left and right subtrees, respectively. The standard interval tree, however, does not well manage typed intervals because of the following reasons.

- By observation, we find that there is a large number of nodes at the bottom level only containing a few intervals (sometimes only one). That means, we require many nodes but only store a small number of intervals.
- The standard structure uses two sorted lists to maintain intervals. To determine intervals intersecting a query point (interval), we need to scan the list until the position after which intervals cannot be the result. The complexity is proportional to the number of intervals intersecting the query. Given a large dataset, too many intervals may be accessed, but the query only needs k intervals.
- The standard structure is not capable of managing types. Therefore, the intervals are iteratively evaluated on the type condition, decreasing the query efficiency.

To overcome these shortcomings, we build the structure in a compact way by defining a bound to determine the minimum number of intervals maintained in a node. If the number of intervals is less than the bound, we stop partitioning intervals which will result in creating nodes for intervals at lower level, and just use one node to store the intervals. A list is defined in the node to maintain the intervals. If the number of intervals is larger than the bound, we propose a new structure to substitute the sorted list for the interval management. The idea is, the interval data space in the node is partitioned into a set of equal-length slots. Two tables are defined in which each row corresponds to a slot and stores a list of intervals. One table maintains intervals *containing* the slots, called *full table*, and the other maintains intervals *intersecting* (partially overlapping) the slots, called *partial table*. Given a query, we calculate its slot and then access tables to retrieve intervals. Since the intervals in the full table contain the slot, we can skip testing the intersection condition, reducing CPU time and I/O accesses. In contrast, intervals in the partial table have to be iteratively evaluated.

Intuitively, the more intervals in the full table, the better the performance is. This is affected by the slot length. A short slot is more likely to be contained by intervals than a long slot. Therefore, it is possible to create more slots with small lengths to increase the number of intervals in the full table. However, this raises two issues. First, the storage overhead increases because an interval will be distributed in all slots that the interval contains. Second, if the query is an interval, a set of slots will be determined. We have to access tables for each slot to retrieve intervals. To sum up, short slots have a high probability of being contained by intervals, but increase the number of table accesses, incurring more I/O cost. This complicates the partitioning issue.

We provide a thorough analysis on the partition strategy and analytically determine the slot length to perform an optimal partition. This enables the structure to be *adaptive* and *self-adjusting* because each node automatically determines the slot length according to the intervals in the node. The slot length differs from node to node, rather than being a dominating value for all nodes. We build type indexes to efficiently find intervals according to types and order intervals on weights to avoid accessing intervals with small weights that cannot contribute to the result. We make the following contributions in the paper:

- We formalize three kinds of top-k queries on typed intervals.
- We build the interval tree in a compact way and propose a new secondary structure to efficiently manage intervals by performing an optimal partition and building type indexes. The index storage cost is analyzed. We also discuss how to update the structure for new arrival intervals.
- We develop efficient algorithms for point and interval queries. To optimize the query procedure, interval queries are converted to point queries. The query time complexity is analyzed.
- For continuous queries, we develop an auxiliary structure to maintain top-k intervals at each time point and propose a heuristic to prune intervals in batch. The structure is general that can also be used to process continuous queries on standard intervals.
- We implement the proposals in a prototype database system and conduct extensive experiments on large real and synthetic datasets to demonstrate the performance advantage of our method over alternative methods.
- The discussion on the generality of the method and the implementation/integration in a conventional database system is provided.

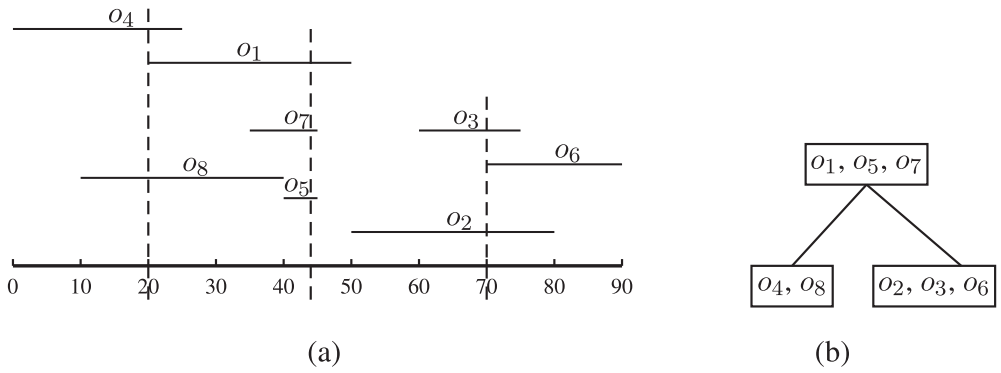


Fig. 2. The interval tree.

The rest of the paper is organized as follows. Section 2 defines the problem and reviews the interval tree. Section 3 details the hybrid index for typed intervals. Section 4.2 presents the algorithms for top-k point and interval queries, and analyzes the time complexity. Section 5 addresses continuous queries. Section 6 reports the results of the experimental evaluation. Section 7 provides the discussion. Section 8 reviews the related work, followed by conclusion in Section 9.

2. Preliminaries

2.1. Problem definition

Let the database O be a set of objects, each of which represents a typed interval. Interval start and end points are defined in a real domain. The type domain is a set of positive integers, denoted by T, and the weight domain is a set of positive real numbers, denoted by $\mathbb{R}^+$.

Definition 2.1. Typed intervals
$I = \{(s, e)|s, e \in \mathbb{R}, s < e\}$
$O = \{(i, t, w)|i \in I, t \in T, w \in \mathbb{R}^+\}$

We associate a type with each interval to enrich the data representation and support applications involving diverse intervals or intervals coming from different data sources. Standard intervals (intervals with the same type) are a special form of typed intervals, i.e., $|T| = 1$. We investigate three kinds of top-k queries, formulated by:

Definition 2.2. Top-k point queries
Given a query $Q(x, t, k)$ in which $x \in \mathbb{R}, t \in T$, and k is a positive integer, the system returns k tuples $O' \subseteq O$ decreasingly ordered by weight such that $\forall o' \in O'$:
(i) $Q.x \in o'.i$;
(ii) $Q.t = o'.t$;
(iii) $\nexists\, o \in O \setminus O'$: $Q.x \in o.i \wedge Q.t = o.t \wedge o.w > o'.w$.

Likewise, we define a top-k interval query $Q(i, t, k)$ ($i \in I$) by changing the condition (i) to $Q.i \cap o'.i$. Furthermore, we have

Definition 2.3. Top-k continuous queries
Given a query $Q_c(i, t, k)$, at each time point $x \in Q_c.i$ the system returns a set of k tuples $O' \subseteq O$ decreasingly ordered by weight such that $\forall o' \in O'$:
(i) $x \in o'.i$;
(ii) $Q_c.t = o'.t$;
(iii) $\nexists\, o \in O \setminus O'$: $x \in o.i \wedge Q_c.t = o.t \wedge o.w > o'.w$.

2.2. Interval tree

Our method is based on the standard interval tree [14]. Using the running example, we review the structure, as demonstrated in Fig. 2. Initially, all intervals are sorted in ascending order according to start and end points. The structure is built recursively from the root down following the procedure: Step 1, a split point among all intervals is computed, denoted by p. Step 2, we use p to divide the interval set into three parts: (i) intervals fully to the left of p; (ii) intervals containing p; (iii) intervals fully to the right of p. The split point should be picked in such a way that the tree is relatively balanced, usually the center point. Step 3, a node is created to hold the part (ii) and two child pointers are defined for nodes maintaining the parts (i) and (iii), respectively. We repeat Steps 1–3 for (i) and (iii) until no interval is left. The *primary structure* is a complete binary tree and each internal node is associated with two lists of intervals that form the *secondary structure*.

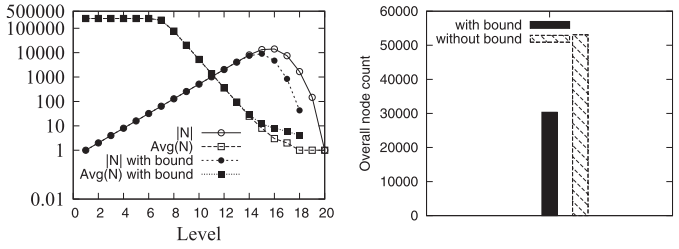
An interval tree node contains three parts: (i) the split point p; (ii) left and right node pointers; and (iii) left and right lists holding intervals containing p. Intervals in the left and right lists are sorted by start and end points, respectively. During the query procedure, the algorithm traverses a path from the root node to the bottom level and progressively reports intervals intersecting the query. Given a node, if the query is smaller than the split point, we walk through the left list to report intervals, and then call the left tree. Otherwise, we walk through the right list and call the right tree. This procedure is iteratively executed until no node is found. Since intervals in the left and right lists are sorted, the search procedure terminates when the visited interval does not intersect the query.

The storage cost is $O(n)$ because each interval is only stored at one node and exactly twice: once in the left list and the other in the right list. An interval tree is a balanced binary search tree and has the depth $O(\log n)$ [33].

3. Hybrid representation

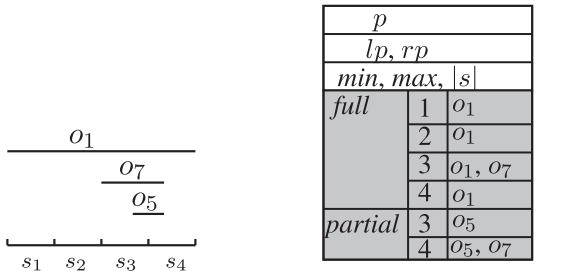
3.1. Motivation

According to the standard algorithm of creating an interval tree, the procedure stops partitioning the interval set until no interval is left. By observation, we find that the number of intervals in a node decreases when the level of the node increases, assuming the level of the root node is 1. Usually, the interval count becomes a small value (sometimes only one) for the majority of nodes at the bottom level. Given a set of intervals, we store intervals intersecting the split point in a node and process the rest of the intervals at the next level. Since the tree is created following a top-down approach, the number of intervals decreases gradually from the root to bottom level. The recursive procedure to build the index termi-



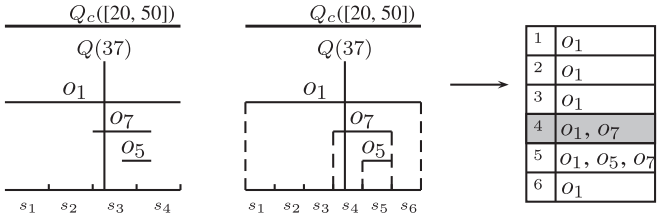
(a) $|N|$ and $Avg(N)$ (b) Overall node count

Fig. 3. Define a bound.



(a) Partition (b) Full and partial tables

Fig. 4. The new structure.



(a) false hits, $|s| = 4$ (b) fine partition, $|s| = 6$

Fig. 5. The partition impact.

nates if either a branch of intervals all intersect the split point or only one interval is left (no partition is needed).

At each level in the tree, let $|N|$ and $Avg(N)$ denote the node count and the average number of intervals in a node, respectively. For example, in Fig. 2 at level 2 we have $|N| = 2$ and $Avg(N) = \frac{5}{2} = 2.5$. Fig. 3 shows the distributions of $|N|$ and $Avg(N)$ by building an interval tree on 50 million randomly generated intervals over the space [1, 100000]. The results show that $Avg(N)$ is quite small (almost 1) from level 15. The value $|N|$ keeps increasing from level 1 until its largest value and then drops quickly. We can see that at certain levels the method requires a large number of nodes but each only maintains a few intervals.

Since a node corresponds to a record, accessing nodes incurs the I/O cost. Hence, we reduce the number of nodes by defining a bound in order to determine whether the interval set is further partitioned or not. Such a value representing the minimum number of intervals in a node is defined by the record size in the system. If the number of intervals is less than the bound, we do not divide the interval set into three parts but store all intervals in this node, i.e., terminating the recursive procedure. This leads to a compact structure. Using the method, we create the interval tree on the same dataset and report $|N|$ and $Avg(N)$ with the bound in Fig. 3. The tree height decreases from 20 to 18 and the number of nodes is almost reduced by half . According to Kriegel et al. [28], the tree height depends on interval bounds (start and end points). If the data range is large, a high tree will be created.

Intervals in the node created with respect to the bound may not have a common point and thus the split point is not defined. We propose a new structure to substitute the sorted lists for the nodes created by the standard method (i.e., the number of intervals is larger than the bound and all intervals contain the split point), presented in the following.

3.2. Partitioning the space into slots

Given a node, let *min* and *max* denote the minimum and maximum endpoints of all intervals, respectively. We partition the data space [*min, max*] into a set of equal-length slots. Each slot has an unique id numbered from 1 to $|s|$ and represents a subinterval with the length $len = \frac{max-min}{|s|}$. Given a slot s_i ($i \in [1, |s|]$), the space covered by the slot is $[min + (i-1) \cdot len, min + i \cdot len]$. Fig. 4(a) shows the partition for the root node holding $\{o_1, o_5, o_7\}$ by setting $|s| = 4$.

We use slot tables to maintain intervals instead of sorted lists. Each row in the table has a slot id and a list of tuple ids for intervals. Depending on whether the interval contains (i.e., completely cover the slot) or intersects (partially overlap) a slot, we define two slot tables named *full* and *partial*, respectively. In a full table, only intervals containing the slot are recorded. In a partial table, intervals intersecting the slot are recorded. Note that if an interval overlaps (contain or intersect) a slot, it will be stored in either the full or partial table, but not in both. The two tables form the secondary structure. As a result, by removing the sorted lists, components of a node become (i) the split point *p*; (ii) left and right pointers, denoted by *lp* and *rp*; (iii) the minimum and maximum endpoints and the number of slots $|s|$; and (iv) full and partial tables. The new structure to maintain intervals $\{o_1, o_5, o_7\}$ is depicted in Fig. 4(b).

Employing the slot method, to find intervals intersecting a query point, we first determine the slot for the query and then access full and partial tables to retrieve intervals. Since the full table stores intervals containing slots, we do not have to evaluate the intersection condition. In contrast, intervals in the partial table have to be iteratively tested.

3.3. Determining the slots

Intervals in a partial table do not contain a slot. To determine intervals intersecting the query, we need to search the table and iteratively perform the intersecting operation between the interval and the query. This will lead to *false hits*. In other words, an interval is accessed but does not contribute to the result. Consider the query $Q(37)$ in Fig. 5(a). The query intersects s_3 and therefore o_5 is fetched from the partial table. However, o_5 does not intersect the query. False hits increase CPU and IO costs because intervals are fetched from the database and then evaluated.

In order to reduce false hits, the number of intervals in the partial table should be minimized. Intuitively, one can set a large $|s|$ to produce many slots with small lengths. Short slots are likely to be contained by intervals. If the partition is fine enough, the partial table can even be empty, as depicted in Fig. 5(b). In this case, there will be no false hit. Let $P(s_i)$ be the number of intervals in the partial table for the slot s_i. We analytically derive the relationship between $|s|$ and $P(s_i)$.

Although interval start and end points are located at different places in a node, all intervals contain the split point and not all of them fully cover the space [*min, max*]. The number of intervals at each point in [*min, max*] changes because intervals start and end at different places. A special case is that all intervals have the same endpoints, leading to a constant interval count. Such a case will not happen for standard intervals because duplicated data will be

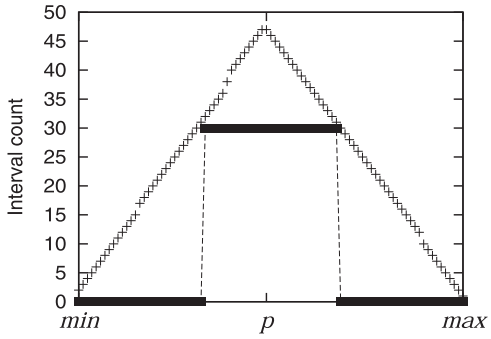


Fig. 6. Interval count distribution.

removed. For typed intervals, they can have the same endpoints but with different types and weights.

Assume start and end points are uniformly distributed over [*min, max*]. The number of intervals in the node reaches the peak around the split point p and decreases from p to *min* and *max*. Fig. 6 depicts the interval count distribution of a node by using example data.

Given a value $x \in [min, max]$, the number of intervals containing x is approximated by:

$$f(x) = \begin{cases} a_1 x + b_1, & x \in [min, p] \\ -a_2 x + b_2, & x \in [p, max] \end{cases} a_1, b_1, a_2, b_2 > 0 \quad (1)$$

Constant values a_1, b_1, a_2, b_2 are determined by *min, max* and the interval count at *p, min* and *max*. One can set $a_1 = a_2 = 0$ and $b_1 = b_2$ for the special case that the interval count is a constant value. Given a slot s_i, the number of intervals in the partial table is calculated by:

$$P(s_i) = \begin{cases} X \in [f(p) - f(s_i.s), f(p)] & \text{if } s_i \text{ contains } p \\ |f(s_i.e) - f(s_i.s)| & \text{else} \end{cases} \quad (2)$$

Here, $s_i.s$ and $s_i.e$ are the start and end points of the slot. If s_i contains p (the split point), all intervals can be in the partial table. Intervals in such a slot are divided into two sets: $O_s = \{o | o.i.s < s_i.s\}$ (intervals whose start points are smaller than $s_i.s$) and $O_l = \{o | o.i.s \geq s_i.s\}$ (intervals whose start points are larger than $s_i.s$). If all intervals in O_s have end points smaller than $s_i.e$, then they are in the partial table. All intervals in O_l are definitely in the partial table. Therefore, $|O_s| + |O_l| = f(p)$. The minimum value appears when intervals in O_s have end points larger than $s_i.e$. In this case, only intervals in O_l are in the partial table, resulting in $P(s_i) = |O_l| = f(p) - f(s_i.s)$. If s_i does not contain p, then $f(x)$ is monotonous for $x \in [s_i.s, s_i.e]$.

The relationship between $|s|$ and $P(s_i)$ is as follows: if $|s|$ increases, then the slot length $len = \frac{max - min}{|s|}$ decreases and so does $P(s_i)$. For example, in Fig. 6, if $|s| = 2$, then $P(s_1) = P(s_2) \approx 45$. If $|s| = 3$, $P(s_1) = P(s_3) \approx 30$ and $P(s_2) \in [15, 45]$.

Now let us consider the number of intervals in a full table.

$$F(s_i) = \begin{cases} f(s_i.s) & s_i.e < p \\ X \in [0, f(s_i.s)] & s_i \text{ contains } p \\ f(s_i.e) & s_i.s > p \end{cases} \quad (3)$$

The relationship between $|s|$ and $F(s_i)$ is as follows: if $|s|$ increases, then $F(s_i)$ increases too. The storage increases because intervals are distributed into slots. The shorter the slots are, the more intervals are stored in each slot. In Fig. 5(a), $\sum_{i=1}^{4} F(s_i) = 5$, while in Fig. 5(b), $\sum_{i=1}^{6} F(s_i) = 6 + 2 + 1 = 9$. The overall storage is $\sum_{i=1}^{|s|} P(s_i) + F(s_i)$ and the value is dominated by $F(s_i)$. This is because an interval is stored two times in the partial table in maximum (two slots that intersect start and end points, respectively) but $\lfloor \frac{|o.i|}{len} \rfloor$ times in the full table. Therefore, the number of false hits and the space cost are inversely related. Furthermore, if $|s|$ increases, more slots will be visited for continuous queries. Consider $Q_c([20, 50])$ in Fig. 5. Four slots are accessed if $|s| = 4$, but six slots are accessed if $|s| = 6$. We need to find an optimal slot setting to achieve the balance.

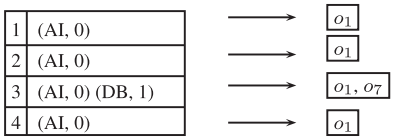


Fig. 7. Type index in the full table.

Consider the lower and upper bounds for the number of slots, that are $[2, max(|s|)]$ in which $max(|s|)$ is the maximum number of slots in a node, determined by the record size. Given a node, let $Avg(|o.i|)$ be the average interval length and the slot count $|s|$ is defined by

$$Min\{\alpha | \frac{max - min}{\alpha} < Avg(|o.i|) \wedge \alpha \in [2, max(|s|)]\} \quad (4)$$

This is motivated by two effects: (i) if $|s|$ increases, then $len = \frac{max - min}{|s|}$ decreases. Short slots increase the probability for intervals containing slots. Meanwhile, we choose the minimal value such that the storage cost is minimized; (ii) different nodes will have their own slot settings because registered intervals result in different *min, max* and $Avg(|o.i|)$ values.

3.4. Managing types and weights

Given a query type and k, the straightforward way to find top-k intervals with a particular type is to iteratively access each tuple and test the type. Then, intervals containing the query type are decreasingly sorted on weights and the first k tuples are returned. To accelerate the query procedure, we manage the intervals in each slot as follows.

In the full table, we do not have to test intervals on the intersection condition. Therefore, intervals are grouped by type and then decreasingly sorted on weights. A type index consisting of a list of items is built, denoted by $I_f = \langle (t_1, off_1), \ldots, (t_{|T|}, off_{|T|}) \rangle$. Each item records a type and the offset for the first interval in the group. Instead of performing a sequential scan, we employ I_f to find the intervals. The search procedure terminates after accessing the first k tuples. Fig. 7 depicts the type index in the full table for $\{o_1, o_5, o_7\}$ by setting $|s| = 4$.

In the partial table, one can employ the same method as in the full table to manage the intervals, i.e., sort intervals by type and weight. However, the search procedure cannot terminate after visiting the first k intervals containing the query type because the first k intervals in the partial table may not all intersect the query. We have to keep searching the intervals and testing the intersection condition until top-k intervals are found in the partial table. To avoid searching intervals that cannot intersect the query, we define an interval that is retrieved by performing the union on intervals with the same type. Start and end points are bounded by the slot. Such a value represents the overall space for intervals with the same type. We add the value into each item of the index. The index for the partial table is denoted by $I_p = \langle (t_1, off_1, i_1), \ldots, (t_{|T|}, off_{|T|}, i_{|T|}) \rangle$, $i_j \subseteq [s_i.s, s_i.e]$, $j \in [1, |T|]$. If the query point is located outside of the bound interval, we can safely prune all intervals in the slot.

Let us consider the node $\{o_2, o_3, o_6\}$ with three slots, as shown in Fig. 8. The added interval for s_1 is [60, 63.3] although $o_3.i = [60, 75]$. The interval for s_2 is the union of o_3 and o_6. Given a query $Q(85, DM, 1)$, the slot s_3 is accessed but o_2 is pruned by the type index because the query is not located in [76.7, 80].

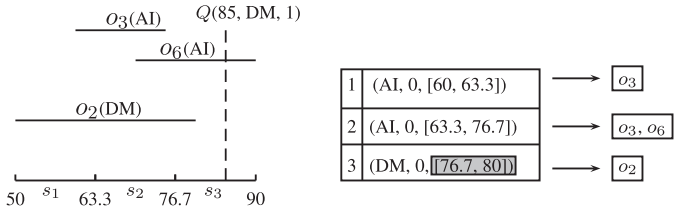


Fig. 8. Type index in the partial table.

3.5. Index storage

We have two ways of managing the intervals in the nodes: using in a list or using the slot representation. If the number of intervals in a node is less than a certain value (the bound in Section 3.1), a list is used. Otherwise, the slot method is applied. Let $n = n_1 + n_2$ be the total number of intervals in which n_1 intervals are stored in the nodes using lists and n_2 intervals are stored in the nodes using slots. The list method needs the storage $O(n_1)$ because each interval is stored once. Consider the storage for the slot representation. Assume m tree nodes are required. In each node, we have full and partial tables, requiring the storage space $\sum_{i=1}^{|s|}(P(s_i) + F(s_i)) + |s| \cdot |T| = O(|s| \cdot (f(p) + |T|))$, where $f(p)$ is the number of intervals in the node and $|T|$ is for the type index in each slot. Let $Max(|s|)$ be the maximum slot count among m nodes. To sum up, the index needs the storage $O(n_1 + Max(|s|) \cdot (n_2 + m \cdot |T|))$ $(n_2 = \sum_{i=1}^{m} f(p))$.

3.6. Update

In addition to maintain the historical data, the index should support updating the incoming data in order to be consistent with the underlying data space. Given a new arrival interval, the index is updated as follows.

We perform a binary search on the primary structure to find the node in which the new interval should be located. If the new interval does not update the minimum and maximum endpoints at this node, the slots for the new interval are determined and then the full and partial tables are updated accordingly. Consider updating the new interval o_9 in Fig. 9(a). The interval intersects two slots s_3 and s_4 in which one belongs to the partial table and the other belongs to the full table. We insert the new interval into each table and update the type index, as demonstrated in Fig. 9(b).

A complex case occurs when the new interval updates the minimum or maximum endpoint, e.g., o_{10} in Fig. 9(a). In this case, the full and partial tables are reset at this node because the slot length and slot count will be changed. In the example, inserting o_{10} will lead to five slots and new slot tables are shown in Fig. 9(c).

If we do not find any node to put the new interval (e.g., the new interval incurs an ongoing expansion of the data space), we create a node to hold the interval and insert the new node into the structure. A rotation may be performed to keep the tree balanced. The rotation will change the parent-child relationship among rotated nodes. After the operation, we need to move intervals among rotated nodes if necessary. This is because each interval is registered at the first node whose split point intersects the interval by performing a top-down traversal.

Given a set of new arrival intervals, iteratively updating the index for each interval is costly. The procedure will perform a binary search and then update slot tables. One tree node will be accessed several times for updating different intervals which are in fact located in the same node. To reduce the overhead, in particular the I/O cost, one can perform the update by bulkload [6,10]. We maintain a buffer for the updated node and read/write the node once to perform the update for a batch of new intervals.

4. Algorithms for point and interval queries

4.1. Point queries

To answer such queries, the root node, the query and a min-heap are taken as input. We start from the root node and perform the traversal in a top-down approach. A min-heap with the size k is used to maintain candidates and the heap is kept updating during the query procedure. If the accessed node maintains intervals by a list (the number of intervals is less than the bound), we iteratively test each interval. Otherwise, we compare the query point $Q.x$ with the minimum and maximum endpoints in this node. If $Q.x$ is located in the data space, we calculate the slot for the query and proceed to access slot tables. A subroutine called *AccessSlot* is invoked in which intervals in the full and partial tables are processed individually. The algorithm named *TopK_Point* is given in Algorithm 1.

Algorithm 1 *TopK_Point*(*N*, *Q*, *H*).

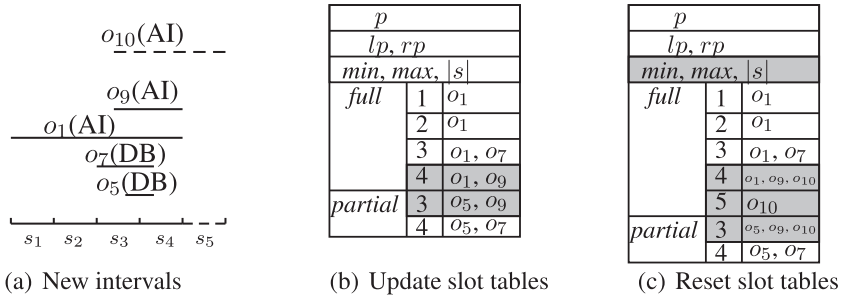
```
1: if N stores intervals by a list then
2:     for all o ∈ N do
3:         if Q.x ∈ o.i ∧ Q.t = o.t then
4:             if |H| < Q.k or Top(H) < o.w then insert o to H;
5: else
6:     if Q.x ∈ [N.min, N.max] then AccessSlot(N, Q, H);
7:     if Q.x < N.p then      TopK_Point(N.lp, Q, H);
8:     else      TopK_Point(N.rp, Q, H);
```

The type index is used to retrieve intervals. In both full and partial tables, intervals having the same type are decreasingly ordered by weights. Intervals in the full table contain $Q.x$, and therefore we do not perform the intersecting testing and just visit the first k intervals. To process intervals in the partial table, we first test whether the interval bound contains $Q.x$. If so, each interval in the table is tested on the intersecting condition. Let $|H|$ be the number of elements in the min-heap. Intervals containing $Q.x$ will be inserted into the min-heap if (i) $|H| < Q.k$ or (ii) $|H| = Q.k$ and their weights are larger than the minimum value in the heap.

Using Q(37, DB, 1) in the running example, we elaborate on the procedure by referring to Fig. 10. Starting from the root node, we calculate the slot for the query, that is s_3 for 37. By accessing the full table, o_1 is pruned and o_7 is put into the heap. In the partial table, although o_5 is a DB project, the interval does not contain the query point and will be discarded. Because of $Q.x < p$ (the split point is 45), the next node to access is $\{o_4, o_8\}$. o_4 does not intersect the query. o_8 is a DB project and contains the query, but its weight is smaller than the minimum value in the heap, i.e., the weight of o_7. Therefore, o_8 will not be inserted into the heap. The query procedure terminates at this node and returns o_7 as the result.

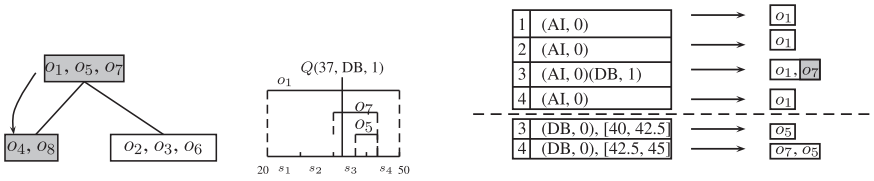
Remove long intervals. In some cases, long intervals may be not interesting, e.g., if they span the overall data space. Users may want to exclude them in the result. To support the operation, the query expression is extended to include predicates on the interval length. Long intervals will be removed when we access the full and partial tables. If the slot length is larger than the query, all intervals in the full table can be removed. Otherwise, the intervals will be checked before putting into the min-heap. Intervals with lengths larger than the defined value will not be considered. In Algorithm 2, evaluating interval lengths occurs after line 1 for full table and after line 5 for partial table.

Time complexity. The time cost consists of two parts: (i) traverse the binary tree and (ii) retrieve intervals from the slot tables and insert them into the min-heap. Given a set of n typed intervals, the first part is $O(\log n)$ as we visit at most one node at each



(a) New intervals (b) Update slot tables (c) Reset slot tables

Fig. 9. Update the structure.

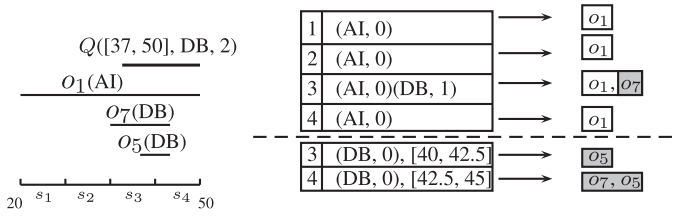


(a) visited nodes (b) root node (c) slot tables for root node

Fig. 10. Example of point query.

Algorithm 2 *AccessSlot(N, Q, H).*

```
1: calculate the slot id and retrieve intervals containing Q.x from full
   and partial tables, denoted by F and P;
2: for all o ∈ {o_1, o_2, ... , o_k} ⊆ F do
3:     if |H| < Q.k ∨ Top(H) < o.w then insert o to H;
4: if the interval bound contains Q.x then
5:     for all o ∈ P do
6:         if Q.x ∈ o.i then
7:             if |H| < Q.k ∨ Top(H) < o.w then
8:                 insert o to H;
```



(a) Slots and query (b) Query processing

Fig. 11. Example of an interval query.

level. Consider the part (ii). Let K be the total number of accessed intervals and we require $O(K \cdot \log k)$ to report top-k intervals.

Theorem 4.1. *The time cost is* $O(\log n + K \cdot \log k)$.

We analyze K as follows. If the visited node maintains intervals by a list, then $O(B)$ intervals are accessed in which B is the minimum number of intervals in a node (the bound defined in Section 3.1). During the query procedure, such a node appears only once (at the bottom level). The other visited nodes define slots. The slot for the query is determined in constant time. Assuming interval types are uniformly distributed, the number of accessed intervals in each node is bounded by $[k, \frac{f(p)}{|T|}]$. The lower bound occurs when the partial table is empty (or all intervals in the partial table have smaller weights than intervals in the full table) and thus only k intervals from the full table are accessed. The upper bound occurs when all intervals containing the query type are accessed. We make use of the Equation (1) to set the value at the split point, i.e., $f(p)$. The average interval count over $[min, max]$ is determined by

$$\frac{\int_{min}^{max} f(x)dx}{max - min} = \frac{\int_{min}^{p}(a_1x + b_1)dx + \int_{p}^{max}(a_2x + b_2)dx}{max - min} \approx \frac{(a_1 - a_2)f^2(p) + (b_1 - b_2)f(p)}{2 \cdot (max - min)} \quad (5)$$

Summarizing all accessed nodes, K is bounded by $[B + k \cdot \log n, \frac{n}{|T|}]$. We do not make any assumption about interval endpoints. If start and end points are represented by integers, the complexity becomes $O(\log n + (B + k \cdot \log n) \cdot \log k)$. As the minimum interval length is 1, one can set the slot length by 1. Then, all slots are contained by intervals, leading to empty partial tables. Therefore, only k intervals in the full table are accessed at each node. The integer representation has some practical applications. In transaction-time databases, time is assumed to be discrete and represented by a succession of non-negative integers [39].

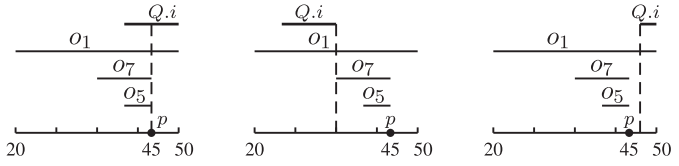
4.2. Extension to interval queries

We can easily adjust the algorithm *TopK_Point* to support interval queries. For each accessed node, a set of slots will be determined for the query. We access full and partial tables for each slot to obtain intervals and insert them into the heap. Consider Q([37, 50], DB, 2) in Fig. 11. Two slots $\{s_3, s_4\}$ are determined for the query in the root node. We iteratively access each slot and have the min-heap $H = \{o_5, o_7\}$ after processing the node.

The shortcoming of the straightforward method is that all slots intersecting the query interval have to be accessed. The complexity is proportional to the length of the query interval. If a long interval is issued, all slots may be visited. To speed up the search process, we propose an optimal method such that the complexity is not effected by the length of the query interval.

Given a query, if $Q.i$ intersects a node, there are two relationships between the split point p and $Q.i$.

- $p \in Q.i$: Note that p is not the middle point of intervals at this node but the split point to partition intervals. Since all intervals at this node contain p, they also intersect $Q.i$. Therefore, the intersecting condition does not have to be evaluated. An example is given in Fig. 12(a).



(a) contain (b) disjoint, $Q.i < p$ (c) disjoint, $Q.i > p$

Fig. 12. *Q.i* and *p*.

- $p \notin Q.i$: There are two possibilities: (i) *Q.i* is located on the left of *p*, i.e., $Q.i.e < p$; and (ii) *Q.i* is located on the right of *p*, i.e., $Q.i.s > p$, as exemplified in Fig. 12(b) and (c), respectively. In case (i), we choose *Q.i.e* as the query and perform a point query. This is because if an interval does not intersect *Q.i.e*, its start point is between *Q.i.e* and *p*, and therefore cannot intersect *Q.i*. In case (ii), we choose *Q.i.s* as the query and perform a point query.

To conclude, we use the following lemma to perform queries.

Lemma 1. *We transform an interval query into a point query in each node by setting*: (*i*) $Q.x = p$ *for* $p \in Q.i$; (*ii*) $Q.x = Q.i.e$ *for* $Q.i.e < p$; *and* (*iii*) $Q.x = Q.i.s$ *for* $Q.i.s > p$.

Proof. We prove Lemma 1 by considering the correctness, no duplicate and no missing result. The first two properties are guaranteed by the point query algorithm and we prove the third in the following:

If $p \in Q.i$, assume there is an interval *o* in the node that $o.i \cap Q.i$ but is not found. Since we use *p* to perform the query, *o.i* is located either on the left or right part of *p*. This contradicts the condition that all intervals in the node contain *p*.

If $p \notin Q.i$, let *o* be the missing interval such that $o.i \cap Q.i$. If $Q.i < p$, we use *Q.i.e* to perform the query and therefore the missing interval *o* is located either on the left or right part of *Q.i.e*. If *o* is on the left part, *o.i* cannot contain *p*. If *o* is on the right part, *o.i* cannot intersect *Q.i*. Both lead to contradiction. The case $Q.i > p$ is similar and the proof is omitted here. □

We transform an interval query to a point query. The method is optimal because we terminate the search if the start point of an interval is larger than the query point. In contrast, given a query interval, we will terminate the search when the start point of an interval is larger than the end point of the query. Our method reduces the number of accessed intervals. Lemma 1 can also be used for intersecting queries on standard intervals.

Algorithm 3 *TopK_Interval(N, Q, H).*

```
1:  if N stores intervals by list then
2:      for all o ∈ N do
3:          if o.i ∩ Q.i ∧ o.t = Q.t then
4:              if |H| < Q.k ∨ Top(H) < o.w then
5:                  insert o to H;
6:  else
7:      if Q.i ∩ [N.min, N.max] then
8:          apply Lemma 1 to perform a point query;
9:      if Q.i.s < N.p then TopK_Interval(N.lp, Q, H);
10:     if Q.i.e > N.p then TopK_Interval(N.rp, Q, H);
```

We give the algorithm in Algorithm 3. Different from a point query, an interval query may incur both left and right child nodes being visited, see lines 9 and 10. Instead of performing $O(\frac{|Q.i|}{len})$ (*len* is the slot length) loops in each node, one time access is sufficient, reducing both CPU and I/O costs.

Time complexity. Given an interval, the procedure traverses the tree to find the nodes intersecting the query. For each accessed node, an interval may cause both child nodes to be visited. This depends on the interval length and the space covered by the node. In an optimal case, the interval regresses to a point, leading to $O(\log n + K \cdot \log k)$. In the worst case, we have $Q.i = [Min(o.i.s), Max(o.i.e)]$ such that all nodes will be accessed. Assuming types are uniformly distributed, $O(\frac{n}{|T|})$ intervals intersect the query and thus the time complexity is

Theorem 4.2. *The time cost is* $O(\frac{n}{|T|} \cdot \log k)$.

5. Top-*k* continuous queries

5.1. Framework

We make use of the well-known *filter-and-refinement* strategy to answer top-*k* continuous queries. Specifically,

Filter. This step traverses the tree to find a set of candidates each of which intersects the query and contains the type. During the procedure, a set of slots will be determined for each accessed node and we search both full and partial tables for each slot.

Refinement. This step iteratively checks each candidate from the filter step. If the candidate belongs to top-*k* intervals, we put it into the result set. Otherwise, the candidate is pruned. Because of the continuous query, the result changes at certain points, complicating the query procedure.

The filter algorithm is straightforward, see Algorithm 4.

Algorithm 4 *Filter(N, Q_c).*

```
1:  C ← ∅;
2:  if N stores intervals by list then
3:      for all o ∈ N do
4:          if Q_c.i ∩ o.i ∧ Q_c.t = o.t then C ← C ∪ {o};
5:  else
6:      if Q_c.i ∩ [N.min, N.max] then
7:          calculate slots S on Q_c.i ∩ [N.min, N.max];
8:          for all s ∈ S do
9:              retrieve intervals containing Q_c.t from full and partial
    tables, denoted by F and P, respectively;
10:             C ← C ∪ F;
11:             for all o ∈ P do
12:                 if o.i ∩ Q_c.i then C ← C ∪ {o};
13:     if Q_c.i.s < N.p then C ← C ∪ Filter(N.lp, Q_c);
14:     if Q_c.i.e > N.p then C ← C ∪ Filter(N.rp, Q_c)
15: return C
```

5.2. The refinement

To continuously report top-*k* intervals, we build a main-memory structure called BD-tree (Bound tree). The BD-tree is in principle a binary tree created on the fly and serves to prune candidates that cannot be the answer and maintain top-*k* intervals at each time point. The structure is progressively built in the refinement by iteratively inserting candidates during which candidates may be kept in the structure or pruned. After processing all candidates, we traverse the BD-tree to report the final result.

Each node in the BD-tree is composed of an interval, the left and right child pointers and a sorted list with *k* elements in maximum. Each element defines a weight and an interval id. Elements have the same start and end points, and are increasingly sorted on interval weights. We denote a BD-tree node by $\beta(i, lp, rp, L)$, $i \in I$, $L = \langle(w_1, oid_1), (w_2, oid_2)... (w_k, oid_k)\rangle$. The BD-tree nodes are

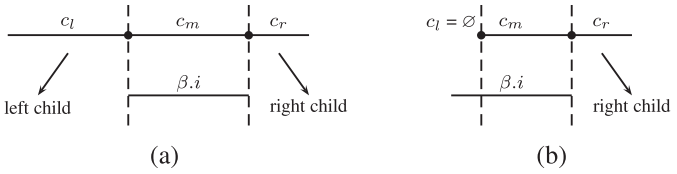


Fig. 13. Split the candidate.

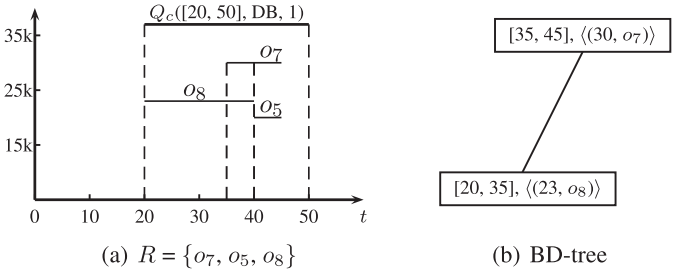


Fig. 14. Example of the refinement.

ordered on interval endpoints and come with the following condition.

Lemma 2. *BD-tree nodes*
Given two BD-tree nodes β_1, β_2: $\beta_1 \neq \beta_2 \Rightarrow \beta_1.i \cap \beta_2.i = \varnothing$.

Nodes should fulfill the condition when they are inserted into the tree. Let $c(i, w, oid)$ be a candidate interval. The type condition is checked in the filter step. Hence, the BD-tree does not evaluate the type predicate. This leads to a general structure such that one can use the BD-tree to answer continuous queries on standard intervals as well. We start from the root node and perform a binary search to find the place to insert c or prune it if there are enough intervals. Given a node β, if $\beta.i \cap c.i$, we will insert c into β. The candidate c will be split into three parts $\{c_l, c_m, c_r\}$ in which they have the same weight but different intervals. In Fig. 13(a), c_l is located on the left part of $\beta.i$; $c_m.i = c.i \cap \beta.i$; c_r is located on the right part of $\beta.i$. Each part is set empty if such an interval does not exist, as demonstrated in Fig. 13(b). We go to the left child of β for c_l and the right child of β for c_r. Regarding c_m, we perform the pruning by using Lemma 3 or update the node.

Lemma 3. *Pruning criteria*
If $|\beta.L| = Q.k \wedge \beta.L[0].w > c_m.w$, *then* c_m *is pruned.*

Proof. Prove by contradiction. Suppose that c_m cannot be pruned, then

case (*i*): $\exists\ \beta.L[i]$, $\beta.L[i].w < c_m.w$. $\beta.L$ is sorted, and this contradicts the condition that $|\beta.L| = Q.k \wedge \beta.L[0].w > c_m.w$.

case (*ii*): $\exists\ \beta'\ (\neq \beta)$, $\beta'.i \cap c_m.i \wedge (|\beta'.L| < Q.k \vee \beta'.L[0] < c_m.w)$. This contradicts Lemma 2. □

Using the running example, Fig. 14 depicts the procedure of inserting the candidates into the BD-tree. The filter step returns $C = \{o_7, o_5, o_8\}$. In the refinement, a root node is initially created for o_7. Then, we insert another interval o_5 in the root node. According to Lemma 3, we prune o_5. Next, o_8 is processed and will be split into two parts: (i) $c_l = ([20, 35], 23, o_8)$); and (ii) $c_m = ([35, 40], 23, o_8)$. For c_l, we go to the left child, which is empty, and create a new node. c_m is pruned according to the pruning criteria. In the end, the query reports {([20, 35], o_8), ([35, 45], o_7)}.

If the pruning condition does not hold, we have to update the node by c_m. There are two cases: (i) $c_m.i = \beta.i$, then we do $\beta.L[0] = (c_m.w, c_m.oid)$ and sort $\beta.L$; (ii) $c_m.i \neq \beta.i$, we split the node. Case (i) is simple. In case (ii), β will be split because $(c_m.w, c_m.oid)$ only corresponds to the interval $c_m.i \subset \beta.i$. Thus, we update the element list for $c_m.i$ but keep the same one for $\beta.i \backslash c_m.i$. There are three split operations depending on where the intersection part $c_m.i \cap \beta.i$ is located, as shown in Fig. 15(a).

- Operation (i). $c_m.i.s = \beta.i.s \wedge c_m.i.e < \beta.i.e$. A new node β_l is created and its content is set as follows: $\beta_l.i \leftarrow c_m.i$, $\beta_l.L \leftarrow \beta.L \cup (c_m.w, c_m.oid)$. Then, we update $\beta.i \leftarrow \beta.i \backslash c_m.i$ and insert β_l into the BD-tree: $\beta_l.lp \leftarrow \beta.lp$, $\beta.lp \leftarrow \beta_l$.
- Operation (ii). $c_m.i.s > \beta.i.s \wedge c_m.i.e < \beta.i.e$. Two new nodes β_l and β_r are created:
  $\beta_l.i \leftarrow [\beta.i.s, c_m.i.s]$, $\beta_l.L \leftarrow \beta.L$;
  $\beta_r.i \leftarrow [c_m.i.e, \beta.i.e]$, $\beta_r.L \leftarrow \beta.L$.
  We update $\beta.i \leftarrow c_m.i$, and insert β_l and β_r into the BD-tree:
  $\beta_l.lp \leftarrow \beta.lp$, $\beta.lp \leftarrow \beta_l$;
  $\beta_r.rp \leftarrow \beta.rp$, $\beta.rp \leftarrow \beta_r$.
- Operation (iii). $c_m.i.s > \beta.i.s \wedge c_m.i.e = \beta.i.e$. A new node β_r is created and its content is set as:
  $\beta_r.i \leftarrow c_m.i$, $\beta_r.L \leftarrow \beta.L \cup (c_m.w, c_m.oid)$.
  Then, we update $\beta.i \leftarrow \beta.i \backslash c_m.i$ and insert β_r into the BD-tree:
  $\beta_r.rp \leftarrow \beta.rp$, $\beta.rp \leftarrow \beta_r$.

Fig. 15(b) shows the example of splitting the node in which we set $Q.k = 2$. The BD-tree already contains ([20, 35], o_8) and ([35, 45], o_7). To insert ([40, 45], o_5), we follow the procedure in case (iii). The node containing ([35, 45], o_7) is split into two parts: [35, 40] and [40, 45]. Since o_7 does not have a right child, ([40, 45], ⟨(30, o_7)⟩) is set as the right child and (15, o_5) is inserted into the list.

The complete algorithm of inserting a candidate into BD-tree is given in Algorithm 5. A candidate is split into several parts and each part is individually processed. We define a Boolean value representing whether the candidate is pruned or not. We mark the value for each part and perform the intersection on all parts to set the final result. The candidate will be pruned if all parts are pruned.

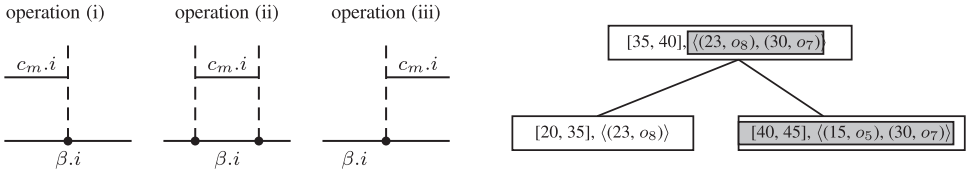
Algorithm 5 *InsertBDTree*(β, c).

```
1: if β = ∅ then
2:     create a node based on c;
3:     return FALSE;
4: split c into {c_l, c_m, c_r};
5: if c_l then
6:     PF_l ← InsertBDTree(β.lp, c_l);
7: if c_r then
8:     PF_r ← InsertBDTree(β.rp, c_r);
9: if c_m then
10:    if c_m is pruned then
11:        PF_m ← TRUE
12:    else
13:        if c_m.i = β.i then β.L[0] ← (c_m.w, c_m.oid)
14:        else split the node;
15:        PF_m ← FALSE
16: return PF_l ∧ PF_r ∧ PF_m;
```

5.3. Enhancing the filter

In the filter step, candidates are processed one by one. The performance can be further improved if the pruning is able to filter a set of candidates once. Recall that we use the full and partial tables to maintain intervals based on the slot representation. For each slot s, we set the candidate from the full table by $c.i = [s.s, s.e]$ and have the following property.

Lemma 4. *Prune in Batch*



(a) Split cases (b) An example

Fig. 15. Split BD-tree node.

Table 1
Dataset statistics.

Name	|O| (million)	[*Min, Max*]	|T|	Weight	Avg(|o.i|)
O1	1	[1, 100000]	100	[1, 500]	500
O2	5				
O3	10				
O4	20				
O5	50				
Bus	25	[1, 96170]	70	[1, 100]	158
Taxi	22	[1, 173459]	8	[0, 255]	47

Table 2
Parameter settings.

|T|	{10, 50, **100**, 200, 500}
|Q.i|, |Q_c.i|	{100, 200, 500, **1000**, 2000, 5000, 10000}
k	{1, 5, **10**, 20, 50}

Given a slot s, let $C(s) = <c_1, \ldots, c_l>$ *be the set of candidates from full table. We have* (*i*) $c_1.w > \ldots > c_m.w$ *and* (*ii*) $\forall\, i, j \in [1, l]$: $c_i.i = c_j.i = s$. *Then, if* $\exists\, i \in [1, l]$: c_i *is pruned, then* $<c_{i+1}, \ldots, c_l>$ *can be pruned.*

Proof. $\forall\, c' \in <c_{i+1}, \ldots, c_l>: c'.i = c_i.i \wedge c'.w < c_i.w \Rightarrow c'$ can be pruned. □

We integrate Lemma 4 into the filter and replace line 10 in Algorithm 4 by Algorithm 6. Since candidates created from the full table have the same interval (see line 1 in Algorithm 6), the path of traversing the BD-tree is the same. We record the path when inserting the first candidate and will follow the path for the remaining candidates. Employing this method, the number of candidates for refinement is reduced. Pruning in batch does not apply to candidates from the partial table because they do not have the same interval.

Algorithm 6 BatchPruning.

```
1: s' ← [s.s, s.e] ∩ Q_c.i;
2: Path ← ∅ and let Flag be the pruning flag;
3: for all o ∈ F do
4:     create a candidate c(s', o.w, o.oid);
5:     if Path then
6:         follow Path to insert c into BD-Tree and set Flag;
7:     else
8:         Flag ← InsertBDTree(β, c) and set Path;
9:     if Flag then break;
```

6. Experimental evaluation

The evaluation is conducted in a standard PC (Intel(R) Core(TM) i7-4770CPU, 3.4 GHz, 8GB memory, 2TB disk) running Ubuntu 14.04 (64 bits, kernel version 4.8.2-19). We develop all index structures and query algorithms (including alternative methods) in C/C++ and integrate the implementation into an extensible database system Secondo [21].

6.1. Datasets and parameters

Both real and synthetic datasets are used in the evaluation. The dataset statistics is reported in Table 1. Synthetic datasets {O1, O2, O3, O4, O5} are generated as follows. The start point of an interval is randomly chosen within the domain [1, 100000], and the length is a stochastic value between 1 and 1000. Types and weights are randomly generated within their domains.

Two real datasets are from a data company DataTang [2]. A sample data can be found at [1]. One is bus card records of Beijing in 2014, named *Bus*, and the other is taxi GPS records of Beijing in 2012, named *Taxi*. In the dataset *Bus*, each tuple stores the number of passengers in a bus during a time interval. The type is the bus route and the weight is the number of passengers. In the dataset *Taxi* (30,000 taxis in total), we take two continuous time points for each taxi and treat them as an interval. Thus, the interval length is the span between two GPS records. The type shows the direction towards the taxi moves to. We partition the direction $\in$ [0, 360) into 8 parts representing northeast, southwest and so on. Each part is defined by an integer between [1, 8]. The speed value is set as weight.

Index creating time and storage. Using the synthetic datasets, we report the time to create our index structure and the storage cost in Fig. 16. We need around 6 s for one million intervals and around 300 s for 50 million intervals. When the data size increases, the time to build the index rises clearly. The storage cost has the same behavior.

Experimental parameters are reported in Table 2 where bold font is for default values. In the evaluation, we use CPU time and I/O accesses as performance metrics and report the result averaged over 30 runs. In the system, we set the record size for holding a node by 1kb, leading to the bound value $B = 19$ (the minimum number of intervals in a node).

6.2. The improvement by bound and slot

We test the effect of defining a bound and performing the partition, and compare the performance with the method that employs a standard interval tree and integrates a bit string into each node to represent all types contained by intervals. For each accessed node, the competing algorithm checks the type and then performs a linear scan in the left or right list. Point and interval queries are evaluated, and datasets O5, *Taxi* and *Bus* are used. Default query parameters are defined.

Fig. 17 reports CPU time and I/O accesses. The results demonstrate that the proposed method achieves more than an order of magnitude performance improvement. In particular, the number of I/O accesses drops significantly. The reason is two-fold: on one hand, defining a bound makes a compact structure and greatly reduces the number of nodes, leading to less number of data blocks; on the other hand, the partition method is employed and at most

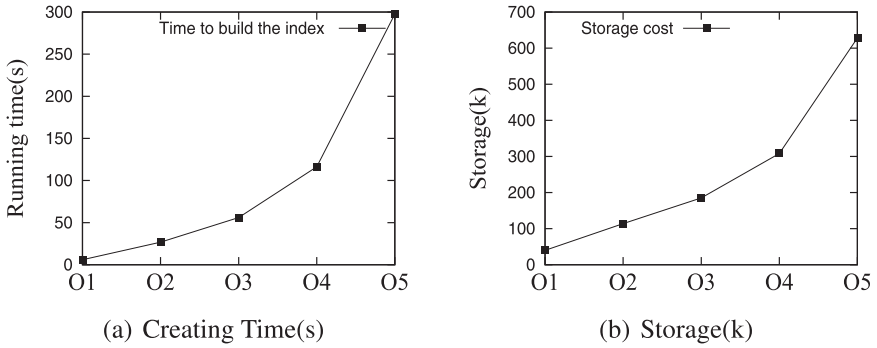


Fig. 16. Index overhead.

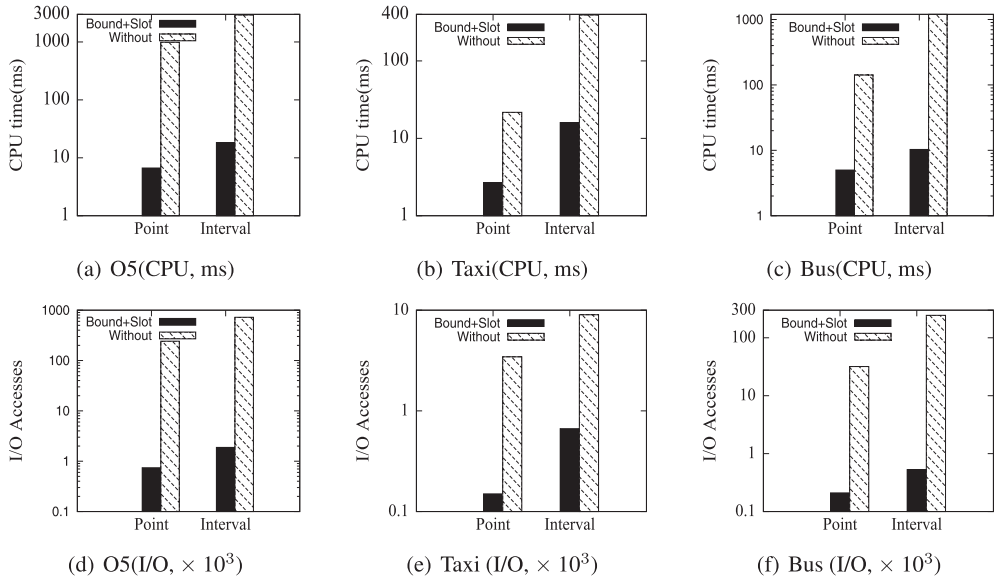


Fig. 17. Effect by Bound + Slot.

k intervals in the full table are accessed in each node. In contrast, the approach without bound and slot needs to process all intervals intersecting the query.

6.3. Verifying the slot setting

We test our slot setting method and show the impact on the performance and storage overhead. We generate four sets of intervals by setting different interval lengths. In each dataset, the start point of an interval is randomly chosen within the domain [1, 100000] and the length value is a random value between [1, 100], [1, 500], [1, 1000] and [1, 2000]. Each dataset contains 50 million intervals.

We compare two slot setting methods: applying the Eq. (4) and defining a fixed value for all nodes. In the system, the size of a record for the slot representation is defined to be 1kb, leading to $Max(|s|) = 19$. The number of slots by our method is different in each node and the value is in [2, 4]. We take the average value to draw the figure. Regarding a fixed slot number for all nodes, we make four values {5, 10, 15, $Max(|s|)$}. Point and continuous queries are evaluated, and default query parameters are used. We report I/O accesses and storage sizes in Fig. 18. The CPU time is less than 15 ms for point queries and varies between 0.5 and 1.6 s for continuous queries. The deviation is not significant and therefore the time cost is omitted. The results demonstrate that (i) both the I/O cost and storage increase when the slot number becomes large (i.e., short slots); (ii) among all settings of interval lengths, our method achieves better performance than defining a fixed value for all nodes; and (iii) long intervals incur less I/O cost for continuous queries because the places where the result changes are fewer.

6.4. Query performance

6.4.1. Alternative methods

The *Baseline* method is to perform a linear scan over the database to check the type and intersection condition. Intervals intersecting the query are put into a min-heap to report top-k results.

Three alternative methods employing index structures are developed. One extends the standard interval tree by integrating a Boolean bit string with the length $|T|$ into each node. Each bit corresponds to a type and represents whether there are intervals with a certain type. If so, the bit is 1. Otherwise, the bit is 0. The secondary structure is defined to be $2 \cdot T'$ ($T' \leq T$) lists. Intervals are partitioned into T' groups according to the type. Each group stores intervals with the same type in the left and right lists. The second method uses a relational interval tree in which the bit string is also integrated. The third method employs a 2D R-tree by treating the interval and the type as two dimensions. The R-tree is built by bulkload. The three methods are named *Ext-I-tree, RI-tree*, and *R-tree*, and our method is named *Slot*. Correspondingly, query algorithms based on those structures are also implemented. We briefly introduce each of them.

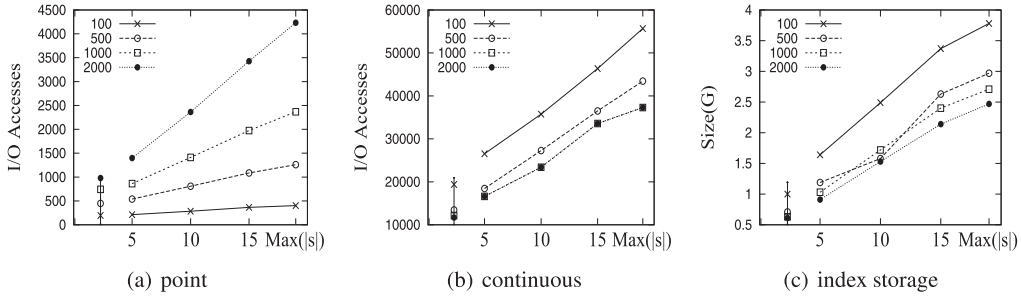


Fig. 18. Test the slot.

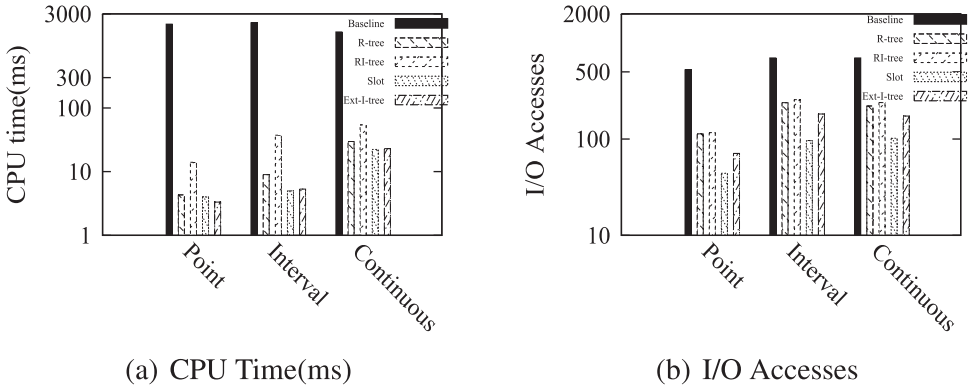


Fig. 19. Using one million typed intervals(O1).

- *Ext-I-tree*: We perform a binary search on the structure during which nodes intersecting the query are accessed. This includes nodes containing the type and intersecting the point (interval). For each accessed node, we retrieve a list of intervals with a particular type and then perform a linear scan to report qualified intervals.
- *RI-tree*: The method performs a binary search on the *primary* structure. For each node intersecting the query, we access the relations to retrieve intervals by employing a B-tree to efficiently find corresponding tuples. Afterwards, each interval is evaluated on the type and intersecting condition.
- *R-tree*: To form the query, we create a box based on the interval and type. The R-tree is traversed in a top-down approach during which nodes intersecting the box are accessed. If the node is a non-leaf node, we iteratively visit each child node. Otherwise, we access each entry in the node to obtain the interval for testing.

In all alternative methods, a min-heap with the size k is used for point and interval queries, and the BD-tree is used for continuous queries. We perform the evaluation by testing (i) the scalability in terms of the number of intervals and types, (ii) the effect of k, (iii) the query length, and (iv) the data interval length. In general, our method requires less CPU time and I/O accesses in most settings. In the case of large type values and a long interval query, *Ext-I-tree* and *R-tree* are sometimes better than our method in terms of the I/O cost.

Effect of Index. The *Baseline* method does not use any index structure. We report the query performance by comparing *Baseline* and methods employing index structures, see Fig. 19. We use the dataset O1 including 1 million typed intervals. The results show that query algorithms using indexes reduce CPU time in two orders of magnitude and I/O accesses at five times compared to *Baseline*. Since the performance advantage of using index structures is obvious, we only compare methods that employ index structures in the following.

6.4.2. Scalability

Using the synthetic datasets, two groups of experiments are carried out to study the effect of scaling the number of intervals $|O|$ and the number of types $|T|$. The results are reported in Figs. 20 and 21, respectively. We perform point, interval and continuous queries. The results demonstrate that *Slot* is superior than alternative methods when scaling $|O|$. We have the following findings when scaling $|T|$.

- If $|T|$ is not large (e.g., < 150), our method outperforms all alternative methods in terms of CPU time and I/O accesses.
- If $|T|$ becomes large, our method is still the best with respect to CPU time. Regarding the I/O cost, the *R-tree* is the best and the *Ext-I-tree* performance also increases. Our method is not very sensitive to $|T|$.

We analyze the results as follows. In the scenario of a large $|T|$, the type predicate is very selective (a good filter). The *2D R-tree* evaluates the type and intersecting predicates simultaneously. The shape of the structure becomes better if the number of types increases. If the type domain is within a small range, this dimension is not comparable to the time dimension, leading to a weak selectivity on the type condition.

Employing the *Ext-I-tree* in which each node maintains $2|T|$ lists, the procedure evaluates the type predicate at first and then collects intervals with a particular type to test the intersecting condition. The number of intervals in each list becomes small when $|T|$ increases, reducing the number of evaluated intervals. In contrast, our method first evaluates the intersecting condition and then determines the intervals with a certain type. If $|T|$ is large, evaluating the type condition before testing the intersecting is better because less candidates are received.

6.4.3. Effect of k

We report the results in Figs. 22–24. Our method outperforms the alternative methods in most cases. For point and interval queries, the performance is not sensitive to k for all meth-

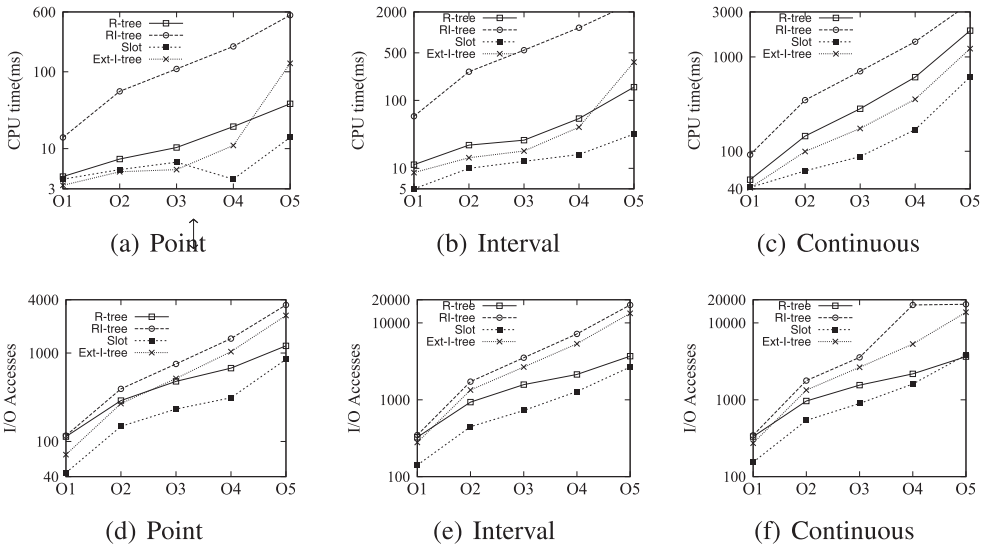


Fig. 20. Scaling $|D|$.

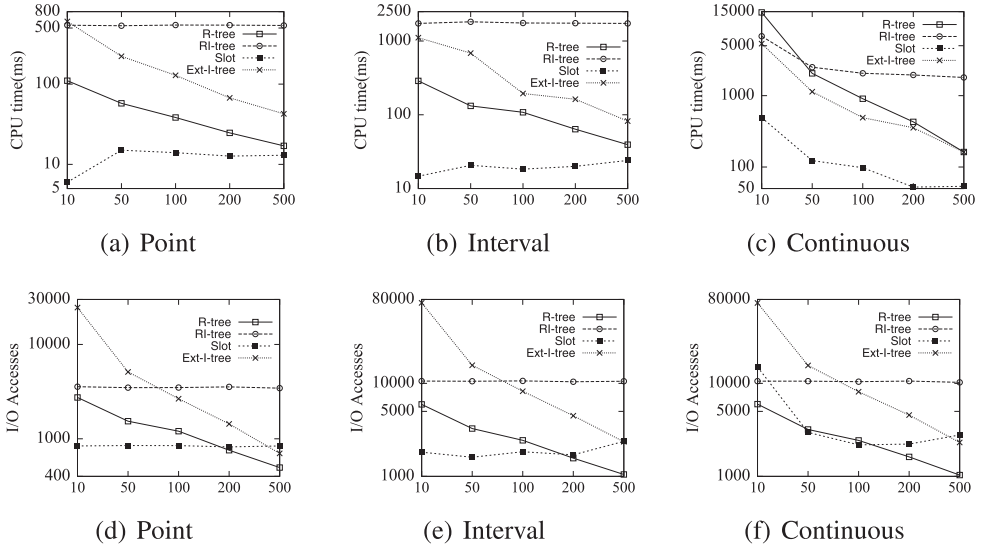


Fig. 21. Scaling $|T|$.

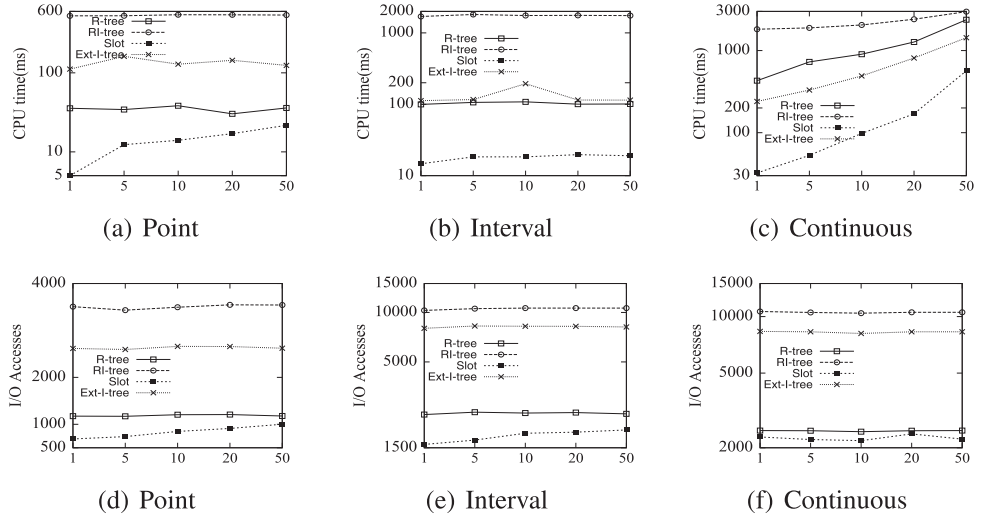


Fig. 22. O5 (k).

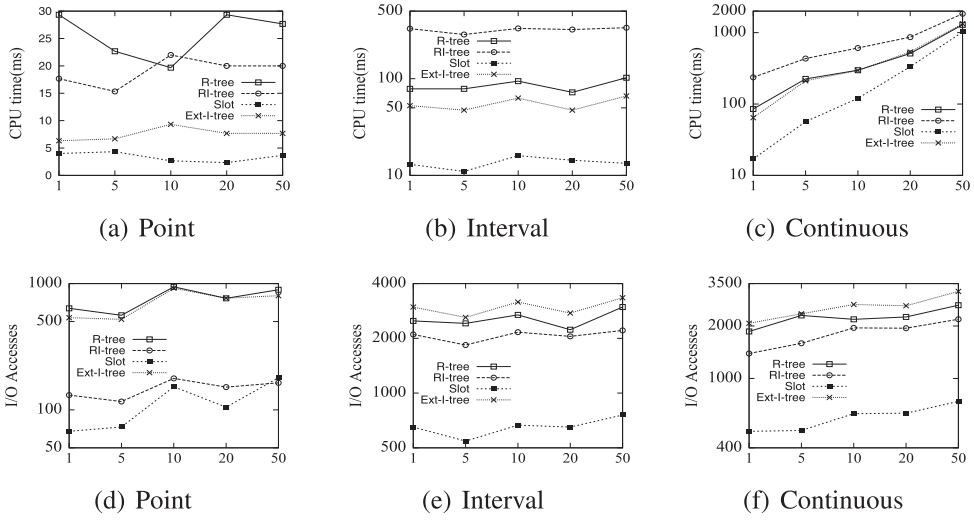


Fig. 23. Taxi (k).

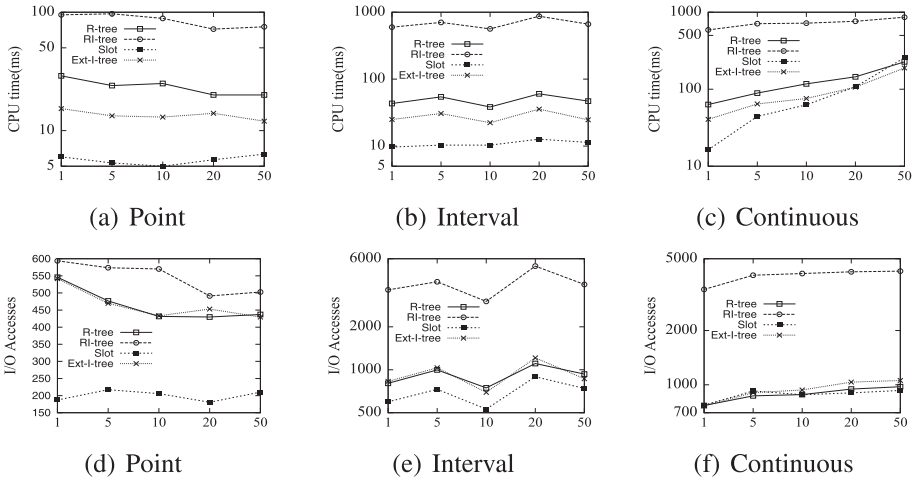


Fig. 24. Bus (k).

ods because the procedure just keeps updating a min-heap, taking $O(\log k)$. The CPU time is a few milliseconds for point queries using *Taxi*, and hence a small deviation may lead to a sharp slope of the curve.

For continuous queries, the CPU time increases proportionately when k becomes large. This is because the procedure continuously updates the results changing over time. The larger k is, the longer the sorted list in the BD-tree is maintained. Whenever a split occurs, the list (main-memory resident) has to be moved from one node to another, requiring the time cost. The *R-tree* and *Ext-I-tree* have slightly less I/O accesses than *Slot* in some settings using *Bus*.

6.4.4. Effect of the query length

We evaluate interval and continuous queries. According to Lemma 1, the query interval is converted to a point. This applies to *Ext-I-tree*, *RI-tree* and *Slot*. The results are reported in Figs. 25 and 26. As expected, when the length of the query increases, more intervals are accessed, incurring both CPU and I/O overhead. Our method achieves the best performance except that the I/O cost is higher than the *R-Tree* and *Ext-I-tree* in a few cases. The reason is, almost all slots in a node are accessed for a long query interval. One interval may be visited several times in different slots, leading to the extra I/O cost.

Table 3
Minimum and maximum lengths.

[1, 100]	[100, 200]	[200, 500]	[500, 1000]	[1000, 2000]

6.4.5. Effect of the data interval length

In this part, we investigate the query performance by scaling interval lengths. Five datasets are used in the evaluation by setting different minimum and maximum lengths, as listed in Table 3.

Each dataset contains 50 million typed intervals, the length of each is randomly selected between the minimum and maximum values. Default query parameters are used. Fig. 27 reports the results of three kinds of queries. The CPU time of our method is not significantly influenced by interval lengths, but I/O accesses increase clearly. This is because the number of slots for an interval increases when the interval length becomes larger, resulting in more times of accessing slots. Our method outperforms the alternative methods in most settings. Similar to the results in other experimental settings, *R-tree* is the most competitive method to *Slot* in terms of I/O accesses but its CPU time is at least 2 times more than ours.

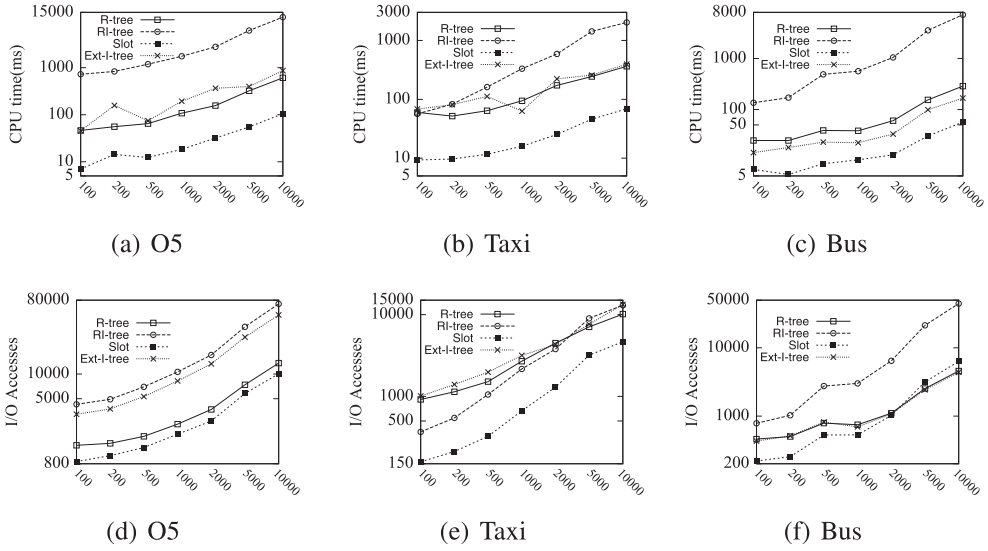


Fig. 25. Interval queries (|$Q.i$|).

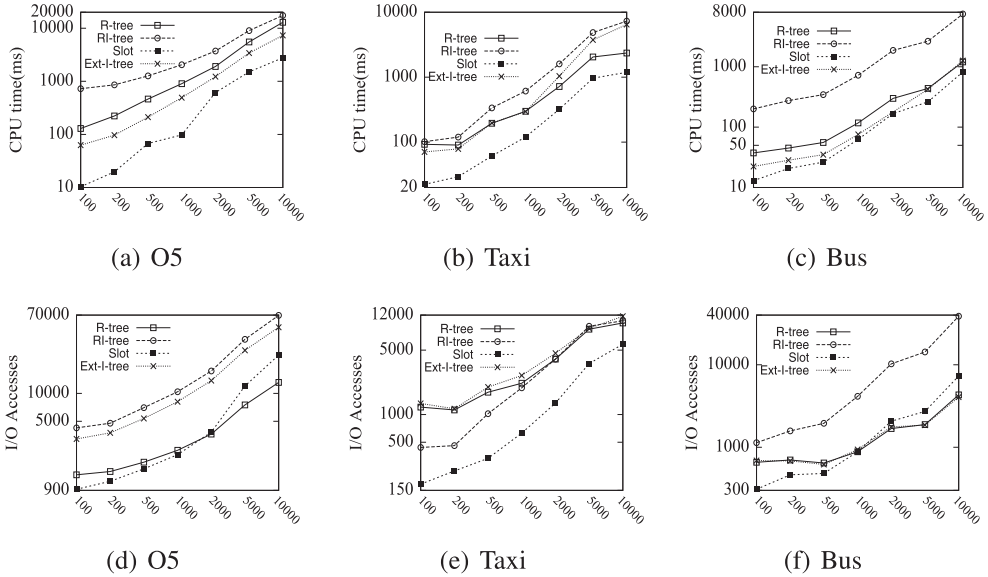


Fig. 26. Continuous queries (|$Q_c.i$|).

7. Discussion

We discuss how to leverage our structure to support a wide range of queries on typed intervals and make a comparison with spatial indexes that are already built in most database systems.

7.1. Predicate and join queries

Specific predicates can be evaluated on intervals. One may want to return intervals with a certain length or define the length of the intersection part between data intervals and the query, e.g., "*return intervals intersecting [10, 100] and the intersection part is longer than 50*". If we use the standard interval tree, the intervals in each node are iteratively evaluated for the predicate. Employing the slot method, we follow the procedure that first determines the corresponding slots for the query. The predicate will be evaluated before accessing the full and partial tables. If the slot length is larger than 50, all intervals in the full table fulfill the condition and do not have to be evaluated. Intervals in the partial table will be iteratively evaluated.

Given two sets of typed intervals, a join query includes a type predicate (t_a, $t_b \in T$) and returns all pairs of overlapping intervals containing query types. For example, return all DB and AI overlapping projects (t_a = DB, t_b = AI). If the two query types are equal, this is the overlapping join on standard intervals. We assume that different query types are issued. The query can be answered by using two proposed indexes built on the dataset.

The idea is as follows. We traverse one index called *target* from top to bottom level by taking each node from the other index called *source*. We first determine whether the interval ranges of the two nodes overlap. If so, we put such a pair of nodes into the candidate set. After iteratively processing each node from the *source*, we receive a group of candidates each of which is in fact a pair of nodes, one from the source and the other from the target.

We find overlapping slots for each candidate and check the type index to determine the intervals containing the query. The intervals are marked *full* or *partial*, representing the table they are from. There are three cases between the intervals from the two slots: (i) full-full; (ii) full-partial; (iii) partial-partial. For case (i), intervals can be directly reported without evaluating the overlap predicate.

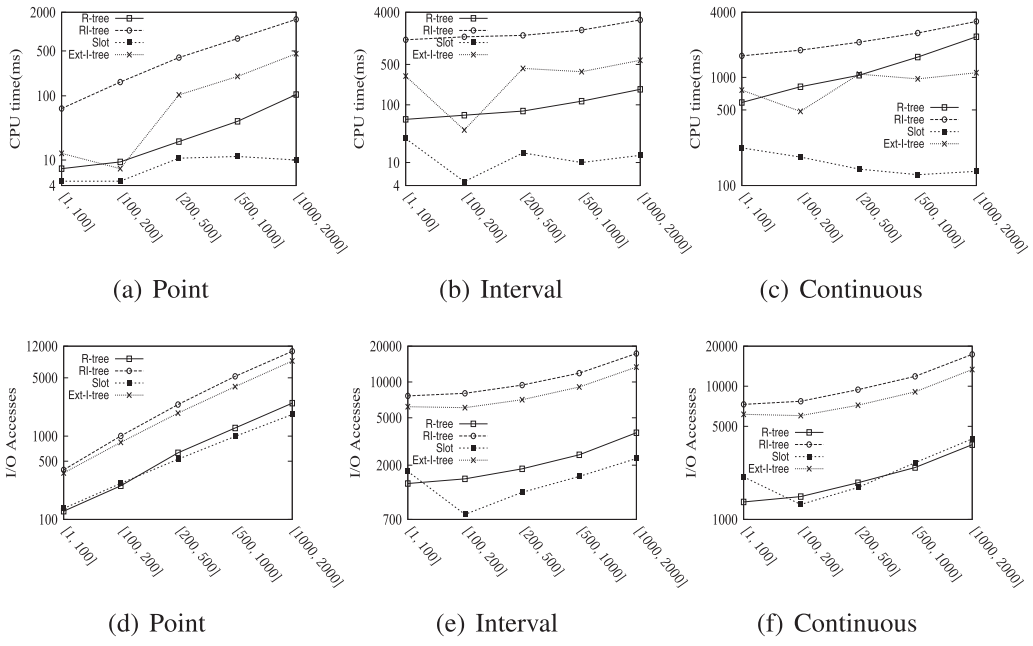


Fig. 27. Scaling interval lengths.

For case (ii), we do not have to fetch intervals from the full table but test whether intervals from the partial table intersect the slot. If so, we report the answer. For case (iii), the intervals are fetched from the two partial tables and then evaluated. The method for evaluating the predicate (e.g., the length) can also be applied to return interval portions with particular lengths.

7.2. Comparison with spatial indexes

In the experimental evaluation (Section 6.4), we compare the performance between the proposed method and the spatial index 2D R-tree. The results show that the R-tree is the most competitive method. Furthermore, such a spatial index has been commonly used in many database systems, making this method attractive. We further analyze the performance difference as follows.

The R-tree treats the data in different dimensions equally and performs the evaluation on all dimensions simultaneously. This leads to a good performance if data is uniformly distributed and the tree has a good shape, i.e., if data in different dimensions is similar in terms of distributions and ranges. However, the domains of different dimensions may deviate significantly depending on the datasets. In such a case, the R-tree may not be the best choice as the index structure. In Section 6.4.2, the R-tree performance is much worse than our method when the number of types becomes small (see Fig. 21). For example, when the number of types is 10, our method achieves an order of magnitude faster in performance than the R-tree. This is because the type domain is within a small range and thus the dimension is not as selective as others. An extreme case occurs when intervals have the same type, leading to 1D R-tree. However, the R-tree is supposed to manage the multidimensional data.

The proposed index in this paper can be implemented in a conventional database system by making use of nested relations. Based on the interval tree in Fig. 2, we exemplify how to integrate the proposed index structure into a conventional database system, as demonstrated in Fig. 28.

The primary structure is represented by a relation in which a tuple corresponds to a tree node and stores basic information as relational attributes such as the split point and the minimum and maximum endpoints. Since we use the full and partial tables to manage typed intervals, two sub-relations are embedded as attributes in the relation. In each sub-relation, a tuple represents a typed interval and stores the slot id. We sort tuples by slot id, type and weight. To efficiently access the full and partial tables, the type indexes are built and also embedded as relational attributes. The indexes essentially record the interval positions in the sub-relations.

8. Related work

8.1. Queries on intervals

Queries on interval data are defined based on the primitive interval relationships. Interval join [20] is studied as a specific join operation in temporal databases. A stabbing-max query [3] returns the interval that contains the query point and has the maximum weight. The sequenced semantics [11,12] provides a relational algebra solution to support outer joins, antijoins and aggregations with predicates and functions over interval timestamped data. The binary interval search [29] counts the intersections between two sets of intervals. The temporal aggregations [35,40] compute aggregates on all tuples whose valid intervals overlap a time interval or point. Interval queries are also investigated in cloud key-value stores [37]. A system named TemProRA is developed to provide the analysis of the top-k temporal probabilistic results of a query on time and probability [36]. This paper differs from those works in that the intervals are associated with categorical types as well as numeric weights. On one hand, our data representation is general, supporting typed intervals and standard intervals. We study not only top-k point and interval queries, but also top-k continuous queries, complicating the evaluation. On the other hand, the query procedure needs to be optimized because several predicates are combined. Existing structures have some drawbacks that decrease the performance, as we have discussed in the introduction. Typed intervals are introduced in [25] with a preliminary solution to answer top-k point and interval queries, but how to set an optimal slot is not analytically provided and continuous queries are not supported.

The closest work to this paper is the top-k interval keyword query [31] in which each interval contains a list of keywords. The goal is to return a set of temporal-textual objects in descending

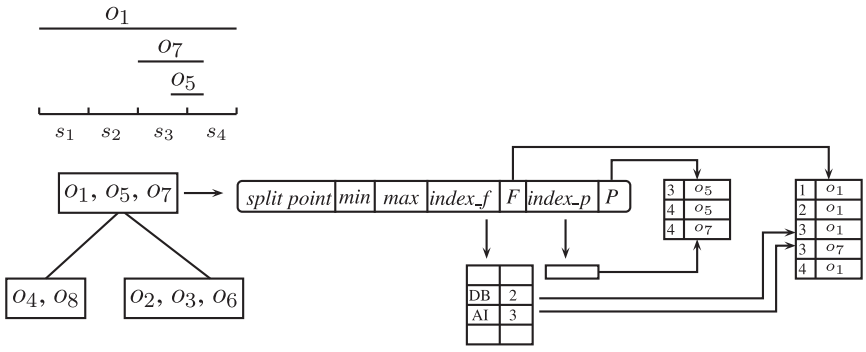


Fig. 28. Integrate the proposed index into a conventional database system.

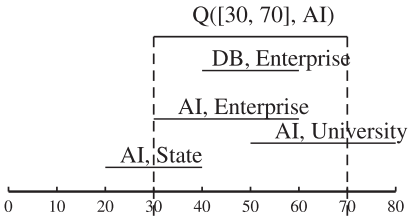


Fig. 29. top-k interval keyword search.

order of relevance scores that combine interval overlap and keyword similarity. Consider the following temporal-textual objects: {([20, 40], (AI, State)), ([50, 80], (AI, University)), ([40, 60], (DB, Enterprise)), ([30, 60], (AI, Enterprise))}, as depicted in Fig. 29. The first word in each object describes the project topic and the second indicates the sponsor. An example of top-k interval keyword queries is "*find the most relevant object that contains AI and exists between* [*30, 70*]". The result is ([30, 60], (AI, Enterprise)) because this object contains the keyword *"AI"* and has the longest interval intersection.

There are several differences between top-k keyword queries and our queries. Firstly, top-k keyword queries only support interval queries because the query computes the overlap between an object interval and a query interval, whereas our top-k queries accept both intervals and points. Secondly, the top-k keyword query evaluates the textual similarity, whereas our query finds objects with the exact keyword match. Thirdly, the concrete algorithms and experimental evaluation are not given for top-k keyword queries.

8.2. Storing and indexing intervals

To facilitate the overlap interval join [20], the interval partitioning [13] divides the interval range of a relation into granules in equal size and uses sequences of granules as partitions of different sizes such that each tuple, associated with an interval, is stored in the smallest covering partition. Although we also use equally sized partitions called slots, we allow a tuple interval intersecting multiple consecutive slots and aim to make the slots as small as possible such that they can be contained by tuple intervals.

Assuming a tuple is associated with a number of intervals, the approach of optimal splitters partitions the interval domain into a given number of buckets such that the size of the maximum bucket, that contains and intersects the largest number of intervals, is minimized [30]. In contrast, we assume that each tuple is associated with one interval, and distinguish between an interval containing and intersecting a partition. We perform the partition in an optimal way in order to achieve the balance between the number of granules and the storage overhead.

Several data structures have been proposed for managing the intervals in the field of computational geometry. The interval tree [14,15] and the segment tree [7,24] are ordered binary trees. To maintain n intervals, the segment tree's storage cost is $O(n \log n)$ and the interval tree needs $O(n)$. The efficiency of answering intersecting queries is the same for the two structures. The priority search tree [34] is a hybrid of binary search tree and priority queue. The structure can efficiently report points in a range where the query on one dimension is unbounded. Each node maintains two values from points in the subtree. One is the *median* value among x-coordinates and the other is an index to the point with the smallest y-coordinate. The structure is not appropriate for indexing intervals because the start and end values belong to the same dimension and the continuous space between them cannot be represented by points. The interval skip list [23] extends the randomized list and is used to efficiently find the intervals overlapping a particular point in active database rule systems. The R-tree [22] can be used to index intervals in the 1-dimensional case. The B-tree and its variants are also widely used to index temporal databases [16]. Since our method is based on the interval tree, we proceed to focus on the techniques closely related to the interval tree.

Several variations based on the interval tree have been devised by combining auxiliary structures, and some secondary storage structures are proposed when data is too large to fit in main memory. By organizing interval endpoints, the augmented B^+-tree [4] reduces both the storage space and the time complexity for insertions and deletions. The interval B^+-tree [9] is used to index a dynamic set of valid time intervals. A tertiary structure uses a binary tree to manage the indexes in a bucket. The B^+-trees are defined on the start points of the intervals augmented with the maximum endpoint in the internal nodes. The interval B^+-tree efficiently processes queries capturing temporal relationships based on the start points of the intervals.

Extending Edelsbrunner's interval tree [14], the relational interval tree [28] uses two relational indexes to index intervals according to a key, a start point, and an end point. All intervals in the tree nodes are stored in two relations: one is for lower bounds of all intervals and the other is for upper bounds. Interval intersection queries can be efficiently answered. The balanced binary tree is not materialized but exists as a purely virtual structure. The relational interval tree is further used for intersection and join queries [17,18].

The external internal tree [5] is an optimal external memory data structure that facilitates stabbing queries on a set of dynamically intervals. The method makes extensive use of two auxiliary structures. Each internal node is associated with an interval that is the union of intervals in child nodes. Such an interval is divided into a set of small slabs. Left and right slab lists are maintained to store intervals with the left and right endpoints located in each

slab, respectively. Depending on the number of intervals intersecting the slab, either a multislab list or an underflow structure is used to store the intervals. The external internal tree is modified to a linear-size structure to solve the one-dimensional version of the interval-stabbing max problem [3]. The method named segment indexes [27] extends conventional database indexing techniques to improve the search performance of spatial data composed of K-dimensional ($K \geq 1$) intervals.

The proposal in this paper manages the intervals in a way clearly different from existing techniques. A compact structure is developed by defining a bound to determine the minimum number of intervals at a node. Our structure contains less nodes and has a smaller tree height than the standard structure. In each node of our tree, we carefully and optimally design slot tables to categorize intervals into two groups instead of simply using lists. The goal is to maximize the number of intervals in one group at the minimum storage cost such that intervals in the group intersecting the query can be reported without performing the intersecting testing. We associate each group with a type index to speed-up look for intervals with a particular type.

9. Conclusion and future research

In this paper, we study top-k queries on typed intervals. Based on the standard interval tree, a new structure is developed to capture intervals, types, and weights in a hybrid presentation, and partition the interval domain in a query-efficient manner. Employing the proposed structure, query algorithms are developed for static (point and interval) and continuous queries that ask for data objects with qualified intervals and types as well as top weights. Extensive experiments using both real and synthetic data are conducted to evaluate the proposed approach. The experimental results demonstrate the efficiency and scalability of our proposal.

There exists a number of directions for future research. The top-k queries in this paper search for individual data objects with respect to different attributes (interval, type, and weight). Finding groups of objects based on aggregates on a particular attribute is an interesting issue. For example, the academic administration often needs to obtain the statistics about the number of students who attend a particular course in a given semester. When two sets of intervals are available, e.g., from two management departments of a university, it is useful to support joins to capture knowledge that is unavailable in a single dataset. Intervals can be associated with probabilistic distribution functions to capture a range of possible values. It is interesting to investigate queries on probabilistic intervals, e.g., return intervals containing the query type and having the highest probabilities of overlapping the query.

Acknowledgment

This work is supported by the Fundamental Research Funds for the Central Universities, NO. NZ2013306.

References

[1] http://dbgroup.nuaa.edu.cn/jianqiu/.
[2] http://factory.datatang.com/en/ (2016).
[3] P.K. Agarwal, L. Arge, K. Yi, An optimal dynamic interval stabbing-max data structure? in: ACM-SIAM SODA, 2005, pp. 803–812.
[4] C. Ang, K. Tan, The interval b-tree, Inf. Process. Lett. 53 (2) (1995) 85–89.
[5] L. Arge, J.S. Vitter, Optimal external memory interval management, SIAM J. Comput. 32 (6) (2003) 1488–1508.
[6] S. Berchtold, C. Böhm, H. Kriegel, Improving the query performance of high-dimensional index structures by bulk-load operations, in: EDBT, 1998, pp. 216–230.
[7] G. Blankenagel, R.H. Güting, External segment trees, Algorithmica 12 (6) (1994) 498–532.
[8] C. Böhm, G. Klump, H.-P. Kriegel, Xz-ordering: a space-filling curve for objects with spatial extension, in: Advances in Spatial Databases, 6th International Symposium, 1999, pp. 75–90.
[9] T. Bozkaya, Z.M. Özsoyoglu, Indexing valid time intervals, in: Database and Expert Systems Applications, 9th International Conference, DEXA, 1998, pp. 541–550.
[10] J.V. den Bercken, B. Seeger, An evaluation of generic bulk loading techniques, in: VLDB, 2001, pp. 461–470.
[11] A. Dignös, M.H. Böhlen, J. Gamper, Temporal alignment, in: ACM SIGMOD, 2012, pp. 433–444.
[12] A. Dignös, M.H. Böhlen, J. Gamper, Query time scaling of attribute values in interval timestamped databases, in: IEEE ICDE, 2013, pp. 1304–1307.
[13] A. Dignös, M.H. Böhlen, J. Gamper, Overlap interval partition join, in: ACM SIGMOD, 2014, pp. 1459–1470.
[14] H. Edelsbrunner, Dynamic Data Structures for Orthogonal Intersection Queries, Technical Report, Tech. Univ. Graz, Graz, Austria, 1980.
[15] H. Edelsbrunner, A new approach to rectangles intersections, parts I and II., Int. J. Comput. Math. 13 (209–229) (1983).
[16] R. Elmasri, G.T.J. Wuu, Y. Kim, The time index: an access structure for temporal data, in: VLDB, 1990, pp. 1–12.
[17] J. Enderle, M. Hampel, T. Seidl, Joining interval data in relational databases, in: ACM SIGMOD, 2004, pp. 683–694.
[18] J. Enderle, N. Schneider, T. Seidl, Efficiently processing queries on interval-and–value tuples in relational databases, in: VLDB, 2005, pp. 385–396.
[19] C. Faloutsos, S. Roseman, Fractals for secondary key retrieval, in: ACM SIGACT-SIGMOD-SIGART, 1989, pp. 247–252.
[20] D. Gao, C. Jensen, R.T. Snodgrass, M.D. Soo, Join operations in temporal databases, VLDB J. 14 (1) (2005) 2–29.
[21] R.H. Güting, T. Behr, C. Düntgen, SECONDO: a platform for moving objects database research and for publishing and integrating research implementations, IEEE Data Eng. Bull. 33 (2) (2010) 56–63.
[22] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: SIGMOD, 1984, pp. 47–57.
[23] E.N. Hanson, T. Johnson, Selection predicate indexing for active databases using interval skip lists, Inf. Syst. 21 (3) (1996) 269–298.
[24] J.L. Bentley, Algorithms for Klee's Rectangle Problems, Computer Science Department, Carnegie Mellon University, Pittsburgh, 1997.
[25] J. Xu, H. Lu, B. Yao, Indexing and querying a large database of typed intervals, in: EDBT, 2016, pp. 658–659.
[26] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, J.S. Vitter, Indexing for data models with constraints and classes, J. Comput. Syst. Sci. 52 (3) (1996) 589–612.
[27] C.P. Kolovson, M. Stonebraker, Segment indexes: dynamic indexing techniques for multi-dimensional interval data, in: ACM SIGMOD, 1991, pp. 138–147.
[28] H. Kriegel, M. Pötke, T. Seidl, Managing intervals efficiently in object-relational databases, in: VLDB, 2000, pp. 407–418.
[29] R.M. Layer, K. Skadron, G. Robins, I.M. Hall, A.R. Quinlan, Binary interval search: a scalable algorithm for counting interval intersections, Bioinformatics 29 (1) (2013) 1–7.
[30] W. Le, F. Li, Y. Tao, R. Christensen, Optimal splitters for temporal and multi--version databases, in: ACM SIGMOD, 2013, pp. 109–120.
[31] R. Li, X. Zhang, X. Zhou, S. Wang, INK: a cloud-based system for efficient top-k interval keyword search, in: CIKM, 2014, pp. 2003–2005.
[32] D.B. Lomet, M. Hong, R.V. Nehme, R. Zhang, Transaction time indexing with version compression, PVLDB 1 (1) (2008) 870–881.
[33] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry (Algorithms and applications), third ed., Springer-Verlag, 2008.
[34] E.M. McCreight, Priority search trees, SIAM J. Comput. 14 (2) (1985) 257–276.
[35] B. Moon, I. López, V. Immanuel, Scalable algorithms for large temporal aggregation, in: ICDE, 2000, pp. 145–154.
[36] K. Papaioannou, M.H. Böhlen, Temprora: top-k temporal-probabilistic results analysis, in: IEEE, 2016, pp. 1382–1385.
[37] G. Sfakianakis, I. Patlakas, N. Ntarmos, P. Triantafillou, Interval indexing and querying on key-value cloud stores, in: ICDE, 2013, pp. 805–816.
[38] R.T. Snodgrass, I. Ahn, A taxonomy of time in databases, in: ACM SIGMOD, 1985, pp. 236–246.
[39] V.J. Tsotras, N. Kangerlaris, The snapshot index: an i/o-optimal access method for timeslice queries, Inf. Syst. 20 (3) (1995) 237–260.
[40] J. Yang, J. Widom, Incremental computation and maintenance of temporal aggregates, VLDB J. 12 (3) (2003) 262–283.