Poster Paper

Indexing and Querying A Large Database of Typed Intervals

Jianqiu Xu[1], Hua Lu[2], Bin Yao[3]
[1]Nanjing University of Aeronautics and Astronautics, China
[2]Aalborg University, Denmark
[3]Shanghai Jiao Tong University, China
jianqiu@nuaa.edu.cn, luhua@cs.aau.dk, yaobin@cs.sjtu.edu.cn

ABSTRACT

Assume that a database stores a set of intervals associated with types and weights. Typed intervals enrich the data representation and support applications involving different kinds of intervals. Given a query time and type, the system reports k intervals that intersect the time, contain the type and have the largest weight. We develop a new structure to manage typed intervals based on the standard interval tree and propose efficient query algorithms. Experiments with synthetic datasets are conducted to verify the performance advantage of our solution over alternative methods.

1. INTRODUCTION

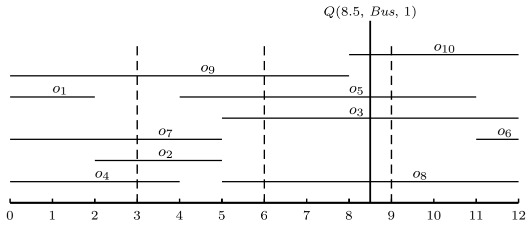
In this paper, we study top-k queries on typed intervals. Assume that a database stores a set of tuples, each of which defines three attributes: an interval with start and end points, a type and a weight. Given a query time and type, the system reports k tuples fulfilling the conditions: (i) intersect the query time; (ii) contain the type; and (iii) have the maximum weight, i.e., return k intervals with maximum weights among all fulfilling conditions (i) and (ii).

To help understand the problem, Figure 1 shows a running example. In traffic monitoring systems, the database stores the number of vehicles appearing in a district over time. There are different kinds of vehicles: {*Taxi*, *Bus*, *Truck*, *Private Car*}. Each tuple records a time interval, the vehicle type and the count. A top-k query is *"return the district with the largest number of buses at the time 8.5"*, and the system returns o_3. The following objects $\{o_3, o_5, o_8, o_{10}\}$ intersect the query time, but only o_3 and o_5 fulfill the type condition.

In the literature, queries on interval data have been studied with operators such as intersecting [3], stabbing [1] and top-k on keyword intervals [5]. However, they do not consider intervals associated with types and therefore do not support applications involving different types of intervals, e.g., various genome intervals in genomic datasets and different versions of data items.

Id	Time	Type	Count
o_1	[0, 2]	*Taxi*	40
o_2	[2, 5]	*Car*	15
o_3	[6, 12]	*Bus*	60
o_4	[0, 4]	*Car*	45
o_5	[4, 11]	*Bus*	20
o_6	[11, 12]	*Truck*	45
o_7	[0, 6]	*Bus*	30
o_8	[6, 12]	*Taxi*	23
o_9	[0, 9]	*Truck*	23
o_{10}	[9, 12]	*Car*	83

(a)



(b)

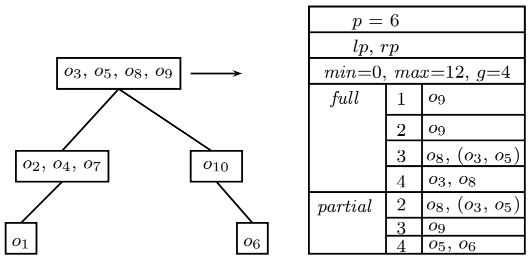
Figure 1: Typed Intervals

Edelsbrunner's interval tree [2] is a popular structure for reporting intervals intersecting a given query. In principle, an interval tree is a binary tree that serves as the *primary* structure. Each node in the tree maintains two lists (*secondary* structure) of sorted intervals. One can directly employ the two-list structure to manage typed intervals, but the method is not optimal. Since the standard interval tree does not support intervals with types, a linear scanning is performed in each accessed node to find intervals that intersect the time and contain the type, even some of them are not equal to the query type. Another problem is, too many intervals are visited. In fact, the query only needs k intervals.

We propose a new structure to replace the sorted lists in each node to maintain typed intervals. Given a node storing a set of intervals, the new method is able to determine part (even all) of the intervals intersecting the query time without accessing the data. An index is built on managing types, leading to quickly finding intervals with a particular type. Employing the new structure, much less intervals are accessed to report k results. We carry out the experimental evaluation to demonstrate the performance of our method by using synthetic datasets.

2. THE SOLUTION

In each node, we replace two lists by a new structure for interval management. The idea is, we partition the interval space into a set of equal-length slots, each of which has a unique id and defines a subspace. We use two tables in which one maintains intervals *containing* the relevant slots and the other maintains intervals *intersecting* the slots, named as *full* and *partial* tables, respectively. Each row in the tables corresponds to a slot and stores a list of interval ids.



$p = 6$		
lp, *rp*		
min=0, *max*=12, *g*=4		
full	1	o_9
	2	o_9
	3	o_8, (o_3, o_5)
	4	o_3, o_8
partial	2	o_8, (o_3, o_5)
	3	o_9
	4	o_5, o_6

Figure 2: Slot representation based on interval tree

Figure 2 depicts the binary tree built on intervals and the slot representation for the root node. The new secondary structure includes the center point p and two child pointers *lp*, *rp*. These items are the same as the traditional structure [2]. Next, *min* and *max* are lower and upper endpoints of all intervals at this node. To perform the partition, the number of slots is defined, denoted by g. In the example, four slots are created for the root node. Intervals located in each slot are first sorted on type and then on weight. The type index is a list of items and each item records the type and the start position of intervals with that type. For example, the index for the third slot in the full table will be {(*Taxi*, 0), (*Bus*, 1)} because o_8 is the first interval in the slot and o_3 is the second interval.

To answer the query, we perform a binary search on the tree. Given a node, we first determine the corresponding slot and then access the full and/or partial tables. Intervals in the full table do not have to be tested on the intersection condition. We use the type index to find intervals having the query type and return the first k intervals. For intervals in the partial table, we find those fulfilling the type condition and then iteratively test each on the intersection condition. Intervals from the two tables are inserted into a min-heap with the size k. We keep updating the min-heap until the searching procedure is terminated. Apparently, the better the performance is, the more intervals are in the full table. This depends on the slot number defined to partition the space.

3. EXPERIMENTAL EVALUATION

We use synthetic datasets in the preliminary evaluation: {*S1*(1M), *S2*(5M), *S3*(10M), *S4*(20M), *S5*(50M)}. The start point of an interval is randomly chosen within the domain [1, 100000], and the length is a random value between 1 and 1000. Let T be the number of types and we set $T = 100$ in the experiment. The weight is randomly selected as an integer from [1, 500].

Three competitive algorithms are developed in the evaluation. One extends the standard interval tree by integrating a boolean bit string representing whether there are intervals with certain types in the node. The secondary structure in each node is defined to be $2{\cdot}T'$ ($T' \leq T$) lists. Each list stores intervals with the same type. The second algorithm uses a relational interval tree [4] in which the bit string is also integrated. The last method employs a 2D R-tree. The three algorithms are named by *Ext-I-tree*, *RI-tree*, and *R-tree*, respectively, and our method is named *Slot*.

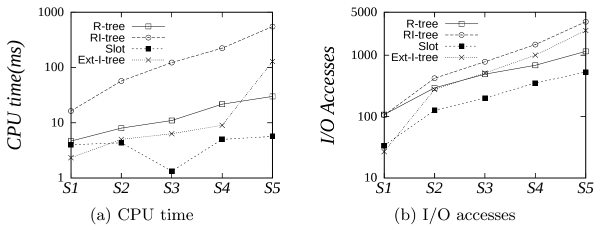


Figure 3: scaling the data size

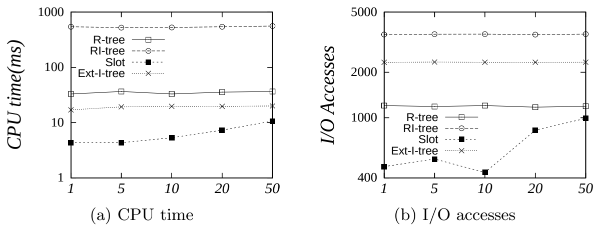


Figure 4: synthetic dataset, *S5*

We perform the evaluation by scaling the number of data intervals and the number of returned intervals k. The CPU time and I/O accesses are reported in Figure 3 and Figure 4. The results demonstrate that our method significantly outperforms other methods, e.g., 3-6 times faster than *R-tree*, 2-10 times faster than *Ext-I-tree*. Since the CPU time is only several milliseconds, a small deviation may lead to a sharp slope of the curve, e.g., in Figure 3(a). The I/O variation in Figure 4(b) is attributed to the randomness of the generated queries.

Acknowledgment

This work is supported by NSFC under grants 61300052 and NSFC of Jiangsu Province under grants BK20130810.

4. REFERENCES

[1] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[2] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Tech. Univ. Graz, Graz, Austria, 1980.

[3] J. Enderle, N. Schneider, and T. Seidl. Efficiently processing queries on interval-and-value tuples in relational databases. In *VLDB*, pages 385–396, 2005.

[4] H.P. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, pages 407–418, 2000.

[5] R. Li, X. Zhang, X. Zhou, and S. Wang. INK: A cloud-based system for efficient top-k interval keyword search. In *CIKM*, pages 2003–2005, 2014.