O2iJoin: An Efficient Index-based Algorithm for Overlap Interval Join

Abstract Time intervals are often associated with tuples to represent their valid time in temporal relations, where overlap join is crucial for various kinds of queries. Many existing overlap join algorithms use indices based on tree structures such as quad-tree, B^+-tree and interval tree. These algorithms usually have high costs since deep path traversals are unavoidable, which makes them not as competitive as data-partition or plane-sweep based algorithms. This paper proposes an efficient overlap join algorithm based on a new 2-layer flat index named as O2i index. The O2i index uses an array to record the end point of intervals and approximates the nesting structures of intervals via two functions in first layer, and the second layer uses inverted lists to trace all intervals satisfying the approximated nesting structures. With the help of the new index, the join algorithm evaluates the join by only visiting the must-be-scanned lists and skipping all others. Analysis and experiments on both real datasets and synthetic datasets show that the proposed algorithm is as competitive as the state-of-the-art algorithms.

Keywords overlap interval join, temporal relation, overlap inverted index, join algorithm

1 Introduction

Temporal relations often model changing states with time-stamping by associating each tuple with a single interval $T = [T.s, T.e]$ to represent the tuple's valid time [1, 2, 3, 4]. For example, the temporal relation R in Fig. 1(a) record the daily salary of each employee working in a labor service company, where T is the period of service time.

To query temporal relations efficiently, a crucial operation is *overlap join*: given two temporal relations R and R', find all pairs of tuples $r \in R$ and $q \in R'$ with overlapping interval, *i.e.,* $r.T \cap q.T \neq \emptyset$. Efficient algorithms for the overlap join provide the query optimizer with more options when other predicates are absent, exhibit a poor selectivity, or must be evaluated after the overlapping interval has been computed [4, 5]. With this comes the need to design effective and efficient algorithms for overlap join operation. And many algorithms have been proposed [1, 2, 3, 4, 9, 10, 11, 12, 19, 20, 22, 23, 24, 26, 31].

Many algorithms among these are index-based. They utilize indices on the join attributes (i.e., endpoints of intervals here) to locate joining tuples. Existing methods have adopted various kinds of tree-structured indices such as quardtree and loose quardtree [29], segment tree, interval tree [8, 28], R-tree [18, 26], $R^\star$-tree [30, 31] and B^+-tree [9, 11, 12, 26]. However, these algorithms usually incur high CPU costs because deep path traversals are unavoidable.

Example 1: The interval tree I of R in Fig. 1(a) are illustrated in Fig. 1(b). Each internal node of I uses a point p to partition the current data subset in three parts. Intervals containing p are indexed in the node itself and intervals before p (after p, *resp.*) are indexed in the left child (right child, *resp.*). The selection of p makes the tree balanced.

When I is used to join a single tuple $\langle q, [3, 9]\rangle$ with R, the root node is visited first. Since $6 \in [3, 9]$, all intervals indexed there are joined with q and then $[3, 9]$ is split into two smaller intervals(*i.e.,* $[3, 5]$ and $[7, 9]$). Then, $[3, 5]$ ($[7, 9]$, *resp.*) are processed recursively in left subtree (right subtree, *resp.*). Thus, the deep paths marked with bold edges are traversed. □

Recently, different algorithms have been proposed to avoid such high CPU costs. Partition-based algorithms [1, 4] divide each input relations into partitions and then join each partition of a relation with some (or all) partitions of the other relation. Inverted indices on endpoints of intervals are integrated into plane-sweep algorithms [3] and forward-sweep algorithms [2] to evaluate overlap join. These algorithms have been shown to be superior compared tree-index-based algorithms.

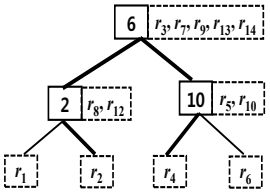
Orthogonal to these work, this paper proposes the O2iJoin algorithm, together with Overlap Interval Inverted Index (O2i Index), to efficiently evaluate the overlap join. The O2i index is a 2-

Footnotes

	T	Employee	Salary
r_1:	[0,1]	Tom	100
r_2:	[3,4]	Bob	100
r_3:	[5,6]	Tom	300
r_4:	[7,8]	Tony	300
r_5:	[9,12]	Jason	200
r_6:	[11,12]	Bill	200
r_7:	[5,10]	Judy	300

	T	Employee	salary
r_8:	[1,3]	Jason	300
r_9:	[1,6]	Tony	200
r_{10}:	[7,10]	Linda	500
r_{11}:	[7,12]	Amily	200
r_{12}:	[0,4]	Judy	200
r_{13}:	[4,12]	Bob	400
r_{14}:	[2,10]	Susan	300

(a) A sample temporal relation R



(b) Interval tree of R

Fig. 1. A simple temporal relation and a tree-structured index

layer flat inverted index, which is built on-the-fly in $O(n \log n)$ time and $O(n \log n)$ space. Via the O2i index, the O2iJoin evaluates the overlap join in $O(n_z + m \log^2 n)$ CPU time by loading each inverted index into memory only once, which makes it as competitive as the state-of-the-art algorithms.

The intuition behind the O2i index is to capture approximately the nesting structures among intervals in one of temporal relations. To do so, all end points (not start points) of intervals in the relation are stored in an sorted array and these points are taken as anchor points. Two functions (*i.e.,* $oiPrev$, $oiSuc$) are defined on the positions of the anchor points to build the approximated nesting structures. For example, Fig. 2(b) illustrates some values of the functions and Fig. 2(a) illustrates the approximated nesting structures on R in Fig. 1(a). To index a tuple r, it is aligned to the "nearest" anchor points first and then it is put into inverted lists (associated with anchor points) in the nesting structures. For example, since r_{14} is aligned to 3 (the 2nd anchor point) and 10 (the 6th anchor point), it is indexed at the 2nd and the 4th ($oiSuc(2)$=4) anchor point. Other intervals in Fig. 1(a) can be indexed similarly to get the O2i index in Fig. 2(c).

To join relation R' with an indexed relation R, O2iJoin join each tuple q in R' with R via the index. To do so, each q is aligned again to the "nearest" anchor points, which locate (with the help of $oiPrev$) "a few" of inverted lists to be visited. For example, when $\langle q, [10, 11]\rangle$ is joined with R in Fig. 1(a) via the index in Fig. 2(c), $[10, 11]$ is aligned to 10 (the 6th anchor point) and 12 (the 7th anchor point). Then, the 7th and the 6th inverted lists are visited. After that, since $oiPrev(6) = 4$ and $oiPrev(4) = 0$, only the 4th inverted list need be scanned to collect all results. In this way, "most" inverted lists are skipped over by O2iJoin. Besides, O2iJoin adopts smart strategies to join all tuples in R' with R concurrently.

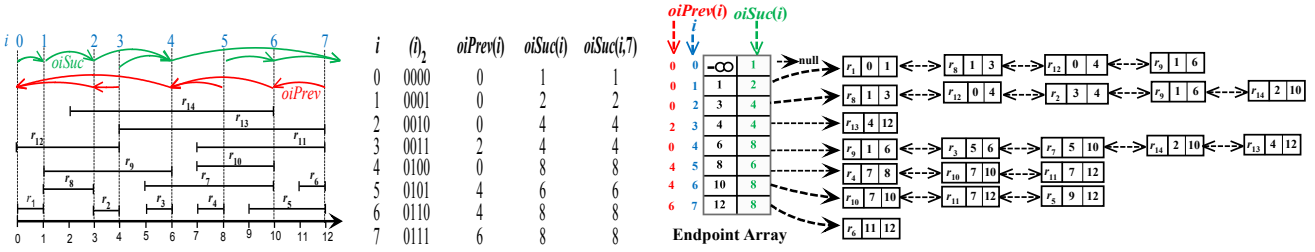
The O2i index is designed as a one-purpose index to support overlap join efficiently. It is built on-the-fly and doesn't support data update. In fact, R's O2i index I changes significantly if a tuple, with its time interval ending at a new point, is inserted. However, it can be adapted to accelerate the join over corresponding data partitions generated by data partirion techniques [4, 11, 12].

Our contributions are summarized below. (1) We propose a flat structure named as O2i index to capture approximately the nesting-structures among intervals. (2) We design an algorithm to build O2i index for any n-tuple temporal relation R in $O(n \log n)$ space and in $O(n \log n)$ time. (3) We propose an algorithm named as O2iJoin to evaluate overlap join efficiently via the O2i index. O2iJoin only visits the must-be-scanned inverted lists in the index and skips over others, which results in a low CPU time. The correctness of O2i is proved. A buffering mechanism is designed to guarantee that each inverted lists of the index is loaded into memory only once. (4) Experiments are conducted on both real datasets and synthetic dataset to compare O2iJoin with the state-of-the-art algorithms. We contend that the O2i index based techniques yielding a promising algorithms for overlap interval join in real-life temporal databases.

The remainder of this paper is organized as follows. Sec. 2 summaries the related work. Sec. 3 is preliminaries. Sec. 4 defines fundamental concepts. The O2i index and its building algorithm are presented in Sec. 5. Join algorithms are developed in Sec. 6. Sec. 7 gives experimental results, followed by the conclusion in Sec. 8.

2 Related Work

Existing algorithms for overlap interval join fall into four basic paradigms: nested-loop, sort-merge, partitioning or index-based. In turn, we



i	$(i)_2$	$oiPrev(i)$	$oiSuc(i)$	$oiSuc(i,7)$
0	0000	0	1	1
1	0001	0	2	2
2	0010	0	4	4
3	0011	2	4	4
4	0100	0	8	8
5	0101	4	6	6
6	0110	4	8	8
7	0111	6	8	8

(a) Nesting structures over R (b) $oiPrev$ & $oiSuc$ (c) the O2i index of R

Fig. 2. Nesting Structures and O2i Index of a simple relation

summary related work along this line.

Nest-loop and sort-merge algorithms. Early nested-loop algorithms [5, 20] are often not competitive due to their quadratic cost, except severe restrictions are imposed like [19] and [20]. Also, traditional Sort-merge based algorithms [19, 21] perform poor(see also the experimental results of [3]), except these special cases such as in append-only databases [22] and for the multi-predicate merge join (MPMG) [23] which supports containment queries on XML data in relation database systems. Recently, data partition techniques are used to avoid backtracking in sort-merge based algorithms [1].

Partition-based algorithms. In [11], the relations are partitioned into non-overlap intervals and partitions are joined from the last to the first one. Histogram is used to accelerate this algorithm [12]. Regions in 2-dimensional space is used to partition data by mapping each interval to a point in the space [13] and Time polygon Index is used to accelerate the join [14].Partition-based algorithms have also been proposed for spatial joins [15, 16, 17, 18]. However, they do not behave as well as interval-specific algorithms in general [5].

Overlap Interval Partitioning (OIP) [4] divides the time range of a relation into some granules and group sequences of contiguous granules to form overlapping partitions such that only the tuples from overlapping partitions have to be joined. OIP was shown to be superior compared to tree-index-based algorithms and sort-merge algorithms [1, 3].

Disjoint Interval Partitioning (DIP) [1] was recently proposed for temporal joins and other sort-based operations(e.g, temporal aggregation). It divides input relations into partitions such that each partition contains only disjoint intervals. Every partition of one input is then joined with all of the other. Since intervals in each partition do not overlap, sort-merge computations can be performed without backtracking. DIP performs well when the input relations consists of short-lived tuples [2], which is consistent with our experiments.

Index-based algorithms. Such algorithms utilize indices on the join attributes (i.e., endpoints of intervals here) to locate joining tuples efficiently. Existing methods have adopted various kinds of tree-structured indices such as quardtree and loose quardtree [27], segment tree, interval tree [6, 26],R-tree [16, 24],$R^\star$-tree [28, 29], B^+-tree [9, 10, 24]. Some researchers used them to support specially temporal join efficient. For example, Time Index [6, 9, 10] was used to evaluate the temporal equijoin, i.e., overlap join with additional equality predicates on a surrogate attribute. MVBT [24, 25] was used to evaluate GTE-Join, i.e., the generalized temporal equijoin which assumes that the key values of the joined tuples lie within a specified range and the time intervals intersect a specified interval. The snapshot index [29] was used as an access method for disk resident transaction-time databases, in which intervals are clustered since database modifications occur in increasing time order. It has been shown that such indices incur high CPU costs when they are used to evaluate overlap interval join [1, 2, 3, 4], because deep path traversals are not avoidable.

Recently, inverted indices on endpoints of intervals are used to support plane-sweep algorithms [3] and forward-plane-sweep algorithms [2]. In these methods, each interval is indexed as its two endpoints in the sorted inverted lists, which can be feeded into plane-sweep algorithms efficiently and provide chances to optimize them via caching and muticore mechanisms. These have been shown to be superior compared to other

methods [2, 3]. The O2i index indexes each interval as a whole and is orthogonal to these methods.

3 Problem Definition

We assume a discrete time domain, denoted by Ω^T, consisting of a linearly ordered set of time points. Each interval consists of a contiguous subset of Ω^T and is represented as an ordered pair $[s, e]$, where $s, e \in \Omega^T$ and $s \leq e$. s (e *resp.*) is called as the inclusive start point (end point *resp.*) of $[s, e]$. For $\forall x \in \Omega^T$, if $s \leq x \leq e$, then we say $x \in [s, e]$. For two intervals $[s_1, e_1]$ and $[s_2, e_2]$, if there is $x \in \Omega^T$ such that $x \in [s_1, e_1] \wedge x \in [s_2, e_2]$, then we say $[s_1, e_1]$ and $[s_2, e_2]$ is overlapped.

In temporal relations, we use tuple timestamp and associate each tuple with a single interval to represent the tuple's valid time. A temporal relation schema is represented as $\mathcal{R} = (A_1, \cdots, A_p, T)$, where A_i is an attribute with domain Ω_i for $i = 1, \cdots, p$ and T is an attribute with domain $\Omega^T \times \Omega^T$. A tuple r over schema $\mathcal{R}$ is a finite set that contains for each A_i a value $v_i \in \Omega_i$ and for T an interval $[r.s, r.e] \in \Omega^T \times \Omega^T$. A temporal relation R over schema $\mathcal{R}$ is a finite set of tuples over $\mathcal{R}$. We say a temporal relation R spans time range $[R.s, R.e]$, where $R.s = \min\{r.s | r \in R\}$ ($R.e = \max\{r.e | r \in R\}$ *resp.*) is the smallest start point (biggest end point *resp.*) of any tuple in R.

Overlap Join. Given temporal relations R and R', the overlap join over R and R' returns the set $\{\langle r, q\rangle | r \in R, q \in R', [r.s, r.e] \cap [q.s, q.e] \neq \emptyset\}$.

Notations in Table 1 is used hereafter.

Table 1. Some Convertions

Notation	Description
R, R'	temporal relations in the overlap join
n, m	size of R and R'
n_z	size of the result of the overlap join
n'	size of $\{r.e \| r \in R\}$
l_{max}	max length of interval lists in O2i index
N	the set of natural numbers
$(i)_2$	the binary representation of $i \in N$

4 Foundation of O2i Index

This section presents fundamental concepts. Two functions $oiPrev$ and $oiSuc$ are defined, their computation are discussed, the nesting structures approximated by them are formalized.

Definition 1: For $\forall i \in N$, its *oi predecessor*, denoted as $oiPrev(i)$, is the non-negative integer obtained by changing the lowest 1-bit of $(i)_2$ into a 0-bit. For convenience, let $oiPrev(0) = 0$. □

Clearly, $oiPrev(i) < i$ for $i > 0$. For $\forall i \in N$, $oiPrev(i)$ can be computed in $O(\log i)$ time. In fact, $oiPrev(i)=(i)_2\&(i-1)_2$, *i.e.,* the bitwise AND operation of i and $i-1$. For example, $oiPrev(6)=6\&5=(110)\&(101)=4$. Therefore, for integers which can be stored in a few of machine words, $oiPrev(i)$ can be computed in $O(1)$ time.

The following lemma describes the nesting structure of $oiPrev$(see Fig. 2(a)).

Lemma 1: $oiPrev(i)<j<i \Rightarrow oiPrev(i) \leq oiPrev(j)$.

Proof: Assume the lowest 1-bit of $(i)_2$ be the kth bit, *i.e.,* $(i)_2=b_1 \cdots b_{k-1}10 \cdots 0$. Then, $(oiPrv(i))_2=b_1 \cdots b_{k-1}00 \cdots 0$. Since $oiPrev(i) < j < i$, we know $(j)_2 = b_1 \cdots b_{k-1}0b_{k+1} \cdots b_m$ and $b_p = 1$ for at least one $p(k < p \leq m)$. If $b_p = 1$ for exactly one p, then $oiPrev(i) = oiPrev(j)$. Otherwise, $oiPrev(i) < oiPrev(j)$. □

Definition 2: For $\forall i \in N$, its *oi successor*, denoted as $oiSuc(i)$, equals $\min\{j | oiPrev(j) < i < j\}$. For convenience, let $oiSuc(0) = 1$. □

The O2i index uses an array to record the end points of all intervals, thus the actual used value of $oiSuc(i)$ will not exceed the length of the array. The definition below formalizes this intuition and makes it possible to store all used values of $oiSuc(i)$ uniformly in a bounded space.

Definition 3: For $i, m \in N$ with $i \leq m$, the *oi successor* of i with respective to m, denoted as $oiSuc(i, m)$, is defined as $oiSuc(i, m) = \min\{oiSuc(i), m + 1\}$. □

For example, Fig. 2(b) illustrates $oiPrev(i)$, $oiSuc(i)$ and $oiSuc(i, 7)$ for $0 \leq i \leq 7$. The nesting structure of $oiSuc(i, m)$ illustrated in Fig. 2(a) can be formalized as the lemma below.

Lemma 2: *For* $\forall i, j, m$ *with* $i<j<m+1$, *if* $j<oiSuc(i, m)$, *then* $i<j<oiSuc(j, m) \leq oiSuc(i, m)$.

Proof: Otherwise, $oiSuc(j, m) > oiSuc(i, m)$ implies $m+1 > oiSuc(i, m) = oiSuc(i)$. Then, according to Def. 2, we know $oiPrev(oiSuc(i)) < i < oiSuc(i)$. Since $i < j < oiSuc(i, m)$, we obtain $oiPrev(oiSuc(i))$

$<i<j<oiSuc(i)$, which implies $oiSuc(j) \leq oiSuc(i) < m+1$ according to Def. 2 again. However, this means $oiSuc(j,m) \leq oiSuc(i,m)$. □

The lemma below gives the complexity to compute $oiSuc(i,m)$ and its proof suggests us an algorithm to compute $oiSuc(i,m)$.

Lemma 3: *For any i, $oiSuc(i)$ can be computed in $O(\log i)$ time. Moreover, for a fixed m, $oiSuc(i,m)$ can be computed in $O(\log m)$ time.*

Proof: Denote $z = \lceil \log_2 i \rceil$ and $j = oiSuc(i)$. We complete the proof by showing the reverse procedure of transferring $(i)_2$ into $(j)_2$.

Assume $(j)_2 = b_1 \cdots b_{k-1}10\cdots0$. It follows Def. 1 that $(oiPrev(j))_2 = b_1 \cdots b_{k-1}00\cdots0$. Since $oiPrev(j) < i < j$, $(i)_2$ must be $b_1 \cdots b_{k-1}0b_{k+1}\cdots b_z$, where $b_l \in 0,1$ for $k+1 \leq l \leq z$ and $b_l = 1$ for at least one $l \in \{k+1, \cdots, z\}$.

Now, we assert that $b_{k+1}\cdots b_z$ is in the form of $1\cdots10\cdots0$, *i.e.,* a continuous sequence of 1-bits followed by some 0-bits. Otherwise, there is a 0-bit followed by an 1-bit in $b_{k+1}\cdots b_z$. Without loss of generality, suppose $b_{k'} = 0$ and $b_{k'+1} = 1$(where $k' \geq k+1$). Thus, $(i)_2 = b_1\cdots b_{k-1}0b_{k+1}\cdots b_{k'-1}01b_{k'+2}\cdots b_z$. Let, $(j')_2 = b_1\cdots b_{k-1}0b_{k+1}\cdots b_{k'-1}10\cdots0$. It is obvious that $j > i$ and $oiPrev(j') < i$. Moreover, $j' < j$, which conflicts with the fact that $j = oiSuc(i)$.

Summarily, if $j=oiSuc(i)$ and $(j)_2 = b_1\cdots b_{k-1}10\cdots0$, then $(i)_2 = b_1\cdots b_{k-1}01\cdots10\cdots0$. In other words, to get $oiSuc(i)$ from i, we only need to reverse the lowest 1-bits and then reverse the next 0-bit, which finishes in $O(\log i)$ time. □

The proof above yields immediately an algorithm(see Alg. 1) to compute $oiSuc(i,m)$ in $O(\log m)$ time. After initialization, the algorithm first skips the lowest contiguous 0-bits (Line 2-Line 4). Then, it reverses the lowest contiguous 1-bits (Line 5-Line 8). After that, the algorithm reverses the next 0-bit(Line 9). At last, it compares the result to $m+1$ and return the minimum value.

Example 2: $oiSuc(5)=oiSuc((101)_2)=(110)_2=6$ and $oiSuc(4)=oiSuc((100)_2)=(1000)_2=8$. Thus, $oiSuc(4,7)=\min\{8,7\}=8$. □

Corollary 4: *$oiSuc(i,m)=i+1$ if $i \leq m$ is an odd.*

The iteration of $oiSuc$ can be used to describe the interaction between $oiPrev$ and $oiSuc$, which is useful in both the index and the join algorithm.

Corollary 5: *Let $i > 0$, $i_1=oiSuc(i,m)$ and $i_2=oiSuc^{(2)}(i,m)$. If $i_1 < j < i_2 \leq m$, then there is $k(>0)$ such that $i_1 = oiPrev^{(k)}(j)$.*

Proof: We consider $(i_1)_2$ and $(j)_2$ again. Since $i_1 \leq m$, we know $i_1 = oiSuc(i)$. It follows from Alg. 1 that the lowest bit of $(i_1)_2$ is a 0-bit. Without loss of generality, assume the lowest k' bits of $(i_1)_2$ be all 0-bits, followed by some continuous 1-bits and then followed by a 0-bit, *i.e.,* $(i_1)_2=b_1\cdots b_{p-1}01\cdots10\cdots0$. Apply Alg. 1 on i_1, we have $(oiSuc(i_1))_2=b_1\cdots b_{p-1}10\cdots00\cdots0$. Since $i_1<j<i_2\leq oiSuc(i_1)$, it must be the case that $(j)_2=b_1\cdots b_{p-1}01\cdots1b_{z-k'+1}\cdots b_z$, and there is at least one 1-bit in $b_{z-k'+1}\cdots b_z$. Let $k = \sum_{z-k'+1}^{z} b_l$ be the number of 1-bits in $b_{z-k'+1}\cdots b_z$. Evidently, we have $k > 0$ and $i_1 = oiPrev^{(k)}(j)$. □

Algorithm 1: oiSuccessor (i,m)

Input: Integers i,m with $i \leq m$
Output: $oiSuc(i,m)$

```
1 If (i =0) Then return 1;
2 u ← i; mask ← 1;
3 while u AND mask = 0 do    // skip 0s
4 |  mask ← mask << 1;       // left shift
5 while u AND mask≠0 do // reverse 1s
6 |  u ← u XOR mask;
7 |  mask ← mask << 1;
8 u ← u XOR mask;  // reverse next 0-bit
9 return min{u, m + 1};
```

The iteration of $oiSuc(i,m)$ over arbitrary i obtains a list of integers, and the length of the list has a closed formula, which provides us a tool to analyse the complexity of algorithms.

Definition 4: For $i,m \in N$ with $i \leq m$, the *oi successor chain* of i with respect to m, denoted as $oiCHAIN(i,m)$, is the integer sequence $i_0<i_1<i_2<\cdots<m+1$, where $i_0 = i$ and $i_k=oiSuc(i_{k-1},m)$ for $k > 0$. □

Lemma 6: *For $i,m \in N (i \leq m)$, $oiCHAIN(i,m)$ contains at most $\lceil \log_2 m \rceil + 1$ integers.*

Proof: Assume $oiCHAIN(i,m)$ be the integer list $(i=)i_0, i_1, \cdots, i_u, m+1$. It follows from Alg. 1 that $(i_{k+1})_2$ $(1 \leq k \leq u-1)$ is obtained by reversing the lowest contiguous 1-bits of $(i_k)_2$ first and then reversing the 0-bit next to them. Let

j_k be the position of the reversed 0-bit in $(i_k)_2$. Then, $j_1 > j_2 > \cdots > j_u$. Furthermore, $(i_0)_2$ contains no 0-bits between positions j_k and j_{k+1} for $1 \leq k \leq u-1$. In other words, 0-bits in $(i_0)_2$ (except the lowest 0-bits) appear only at positions $j_1, j_2, \cdots, j_u$. Since $(i_0)_2$ has at most $\lceil \log_2 m \rceil$ 0-bits, we have $u \leq \lceil \log_2 m \rceil$. □

5 O2i Index

Based on the concepts of $oiPrev$ and $oiSuc$, this section defines the O2i index formally, and then gives an algorithm to build O2i index for temporal relations. We show that the algorithm runs in $O(n \log n)$ space and $O(n \log n)$ time.

Definition 5: Given a temporal relation R spanning time range $[R.s, R.e]$, the O2i index I of R is a two-layer structure. The first layer contains two arrays $I.EA$ and $I.SUC$. $I.EA$ is called as the endpoints array, in which $-\infty$ and all different end points of the intervals of all tuples in R are stored in ascending order. $I.SUC$ is called as the successor array, in which $I.SUC[i]$ stores the *oi successor* of i with respect to the length of $I.EA$. For each unit $I.EA[i]$, the second layer of I contains an inverted list of tuple set $\{r | r \in R, r.e \geq I.EA[i] \geq r.s > I.EA[oiPrev(i)]\}$, which is sorted in ascending order of the end point $r.e$. □

Example 3: For the relation R in Fig. 1(a), its O2i index is illustrated in Fig. 2(c). The endpoint array $I.EA$ stores $\{-\infty, 1, 3, 4, 6, 8, 10, 12\}$ in ascending order, and $I.SUC[i]$=$oiSuc(i, 7)$. $I.List[0]$ is empty. Since $oiPrev(1)$=0, $I.List[1]$ stores tuple set $\{r_1, r_8, r_{12}, r_9\}$ which contains all tuples r satisfying $r.s \geq -\infty$ and $r.e \geq 1$. Similarly, since $oiPrev(2) = 0$, $I.List[2]$ stores all tuples r satisfying $3 \geq r.s > -\infty$ and $r.e \geq 3$, and so on. Obviously, some tuples (*e.g.,* r_{11}, r_{14}) are indexed more than once. □

The nesting structures of $oiSuc$ and $oiPrev$ makes it very simple to build O2i index for a relation R: For $\forall r \in R$, align interval $[r.s, r.e]$ to two positions i_s, i_e ($i_s \leq i_e$) of the endpoint array and add r to the tail of inverted lists $I.List[oisuc^{(k)}(i_s)]$ ($k \geq 0$) till $oisuc^{(k)}(i_s) > i_e$. We first describe this procedure formally and then show its correctness.

Technically, the O2i index of a temporal relation R can be created in three stages (See Alg. 2). The first stage(Line 1-Line 8) initializes the endpoint array. It first sorts all intervals of R in ascending order of their end points(Line 1). Then, it scans the sorted result, and uses each new encountered end point to initialize next unit of the endpoint array $I.EA$(Line 2-Line 7). After that, the length n' of $I.EA$, *i.e.,* the number of different end points in R, is obtained(Line 8). The second stage(Line 9-Line 10) initializes the successor array. it computes $oiSuc(i, n')$ for each $i \leq n'$ and stores them into $I.SUC$ sequentially. The third stage(Line 12-Line 17) scans R again to put each tuple into some inverted lists. Specifically, for each r_j, it first uses binary search to find a position i in $I.EA$ such that $I.EA[i-1] < r_j.s \leq I.EA[i]$ (Line 13), and then stores r_j into inverted lists $I.List[k]$ for $k \in oiCHAIN(i, n')$ and $r_j.e \geq I.EA[k]$(Line 15-Line 17).

Algorithm 2: O2iIndexCreate (R)

Input: An interval Set R
Output: The O2i index I of R

```
1  Sort R such that r_1.e ≤ ... ≤ r_n.e;
2  I.EA[0] ← −∞; I.List[0] ← null;
3  n' ← 1;
4  for j = 1 To n do      // Initialize I.EA
5      if r_j.e > I.EA[n' − 1] then
6          I.EA[n' − 1] ← r_j.e;
7          n' ← n' + 1;
8  n' ← n' − 1; // n' is the # of end points
9  for j = 0 To n' do   // Initialize I.SUC
10     I.SUC[i] ← oiScu(i, n');
11 for j = 1 To n do          // index each r_j
12     i ← binarySearch(r_j.s, I.EA);
       // I.EA[i − 1] < r_j.s ≤ I.EA[i]
13     while i ≤ n' do
14         if r_j.e ≥ I.EA[i] then
15             Add r_j to the end of I.List[i];
16             i ← I.SUC[i];
17         else break;
```

Example 4: For relation R in Fig. 1(a), Alg. 2 first initializes $I.EA$, $I.SUC$ (see Fig. 2). Then,it puts r_j in some ordered lists. For example, when processing r_{14}, since $3 \geq r_{14}.s > 1$, r_{14} is only possible to stored in $I.List[k]$ for $k \in oiCHAIN(2, 7) = \{2, 4\}$. r_{14} is stored in both $I.List[2]$ and $I.List[4]$, because $r_{14}.e > I.EA[2]$ and $r_{14}.e > I.EA[4]$. After all tuples are indexed similarly, the index in Fig. 2(c) is obtained. □

It is notable that Alg. 2 does not refer any *oi predecessor*, although the index is defined via both $oiPrev$ and $oiSuc$. The interaction between $oiPrev$ and $oiSuc$ guarantees that the output is what is desired. The theorem below shows the correctness of Alg. 2.

Theorem 7: *Given an n-tuple temporal relation R, the* O2iIndexCreate *algorithm builds* O2i *index of R in* $O(n \log n)$ *space and* $O(n \log n)$ *time.*

Proof: We first show the correctness, and then analyse its costs.

Let I be the output of Alg. 2. Since all tuples are considered in ascending order of the end points and each tuple are always added at the end of $I.List[i]$, all tuples in $I.List[i]$ are therefore sorted. Thus, we only need to show that for each $i \leq n'$, $I.List[i]$ stores all tuples $r \in R$ satisfying $I.EA[oiPrev(i)] < r.s \leq I.EA[i] \leq r.e$. Assume r is arbitrary such a tuple. We show that r will be indexed in $List[i]$.

Let i_0 be the value obtained in Line 12. It is obvious that $oiPrev(i) < i_0 \leq i$. If $i_0 = i$, then r is added into $I.List[i]$ when Line 13-Line 16 are executed for the first time. Otherwise, $i_0 < i$. Furthermore, we know i_1=$oiSuc(i_0) \leq i$ according to Def. 2. If $i = i_1$, then r is added in $I.List[i]$ when Line 13-Line 16 are executed for the second time. Otherwise, we have $i_0 < i_1 < i$. Let $i_2 = oiSuc(i_1)$. Repeating the procedure above, we obtain a strictly increasing sequence $i_0 < i_1 < i_2 < \cdots$ till i is reached(*i.e.,* $i_k = i$ for some k), in which case r is added into $I.List[i]$ when Line 13-Line 16 are executed for the $k+1$th time.

Next, we analyse the time complexity of Alg. 2. Sorting tuples of R (Line 1) costs $O(n \log n)$ time. The initialization of $I.EA$ (Line 2-Line 8) costs $O(n)$ time. The initialization of $I.SUC$ (Line 9, Line 10) computes $oiSuc(i, n')$ for each $i \leq n'$, and each computation costs $O(\log n')$ time (Lemma 3). Thus, this initialization also costs $O(n \log n)$ time. At last, we analyse the costs to put $r_1, ..., r_n$ into inverted lists(Line 12-Line 17). For each $r_j \in R$, Line 12 costs $O(\log n')$ time. According to Alg. 2, r_j can only be stored into $list[k]$ for $k \in oiCHAIN(i, n')$, which contains at most $\lceil \log n' \rceil + 1$ integers(see Lemma. 6). Since $n' \leq n$, we know r_j is stored in at most $\lceil \log n \rceil + 1$ lists. Thus, putting $r_1, ..., r_n$ into the index costs $O(n \log n)$ time. As a whole, the time complexity of Alg. 2 is $O(n \log n)$.

Finally, we consider the space costs. Clearly, $I.EA$ and $I.SUC$ need $O(n)$ space. Moreover, each $r_j \in R$ is stored in at most $\lceil \log n \rceil + 1$ lists(see last paragraph). Therefore, the space needed by all inverted lists is $O(n \log n)$. Put together, the O2i index of R needs $O(n \log n)$ space. □

Remark. (1) Occasionally, the join algorithms (in Sec. 6) want to find all $r \in List[i]$ with $r.s$ no larger than a given t. To avoid sequential scan, another copy of $List[i]$ in ascending order of $r.s$, which increases the space costs of the index but still in $O(n \log n)$. Moreover, Alg. 2 still runs in $O(n \log n)$ time, if we sort R in ascending order of $r.s$ and execute Line 11-Line 17 of Alg. 2 again.

(2) For large relation R, each inverted list need be written to the disk. To do so, Alg. 2 can assign for each $List[i]$ ($1 \leq i \leq n'$) a buffer of fixed-size. Each time when the buffer is full, the algorithm writes it to the disk and reuses the buffer.

6 Overlap Interval Join with O2i Index

This section develops algorithms to evaluate overlap join via the O2i index. Sec. 6.1 proposes an algorithm to join a single tuple with a relation. Then, Sec. 6.2 adapts it to join two relations.

6.1 Join a single tuple with a relation

The nesting structures captured by the O2i index can help to join a tuple with a relation. First, the tuple can be aligned to the endpoint array and all inverted lists are organized into different groups. Then, the nesting structures are utilized to deal with each group in different manner(*i.e.,* ignored, or sequential scan, or skippingly scan) such that all results are output exactly once. Moreover, when each inverted list is canned, the nesting structures help to decide which part of the list should be processed. Below, we first introduce the alignment and different manners technically, then present the algorithm formally and analyse it.

We join a tuple $\langle q, [q.s, q.e] \rangle$ with a temporal relation R via its O2i index I.

Align tuple to endpoint array. The end point $q.e$ is used to locate a position i_0 in $I.EA$ by invoking binary search such that $I.EA[i_0] \geq q.e >$

$I.EA[i_0-1]$. If i_0 does not exist, then let $i_0 = n'$ (*i.e.,* the length of $I.EA$). Then, set $[n']$ is partitioned into 4 subsets as below.

$A_4 = \{i | I.EA[i] < q.s\}$

$A_3 = \{q.e > I.EA[i] \geq q.s\}$

$A_2 = \{i_0\} = \{i : I.EA[i] \geq q.e > I.EA[i-1]$ *or* $n'\}$

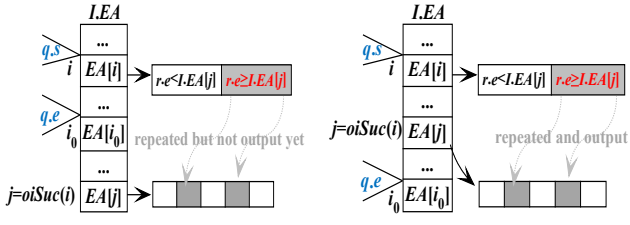
$A_1 = [n'] - A_2 - A_3 - A_4$

Lists $I.List[i]$ in I are grouped according to $i \in A_1, A_2, A_3$, or A_4. Each group will be processed in different manner. A_1 is ignored. $List[i_0]$ is scanned thoroughly to find all $r \in List[i_0]$ satisfying $[r.s, r.e] \cap [q.s, q.e] \neq \emptyset$ (*i.e.,* $r.s < q.e$).

Scan sequentially . A_3 is scanned one by one in descending order. The largest i in A_3 is $i_0 - 1$ if $A_3 \neq \emptyset$. Notice that, for $\forall i \in A_3$, each $r \in List[i]$ satisfies $[r.s, r.e] \cap [q.s, q.e] \neq \emptyset$ and thus $\langle r, q\rangle$ is a result. To avoid producing repeated results, *oiSuc* helps to decide which part of the list should be scanned according to j=$oisuc(i)$>i_0 or not(see Fig. 3).

Case 3a: j=$oisuc(i)$>i_0 (Fig. 3(a)). In this case, $\langle r, q\rangle$ is not output before for $\forall\ r$∈$List[i]$, because $List[j]$ is not processed at all. Thus, $List[i]$ is scanned thoroughly and $\langle r, q\rangle$ is output for $\forall r \in List[i]$.

Case 3b: j=$oisuc(i)$≤i_0 (Fig. 3(b)). In this case, $\langle r, q\rangle$ satisfying $r.e$≥ $I.EA[j] \wedge r$∈$List[i]$ has been output when $List[j]$ is scanned, because such r is also indexed in $List[j]$. Thus, $List[i]$ is only scanned partially and forwardly till $r.e \geq I.EA[j]$. For each encountered r, $\langle r, q\rangle$ is output.



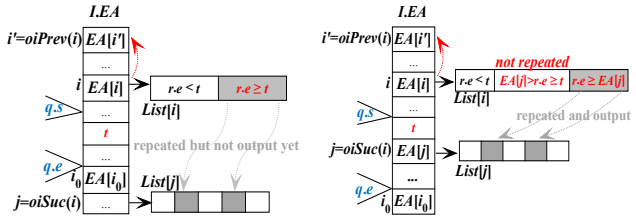
(a) Repeated but not output (b) Repeated and output

Fig. 3. Two case for $i \in A_3$

Scan skippingly. When the smallest i∈A_3 is processed, a time point t=$I.EA[i]$ is obtained. Obviously, $[r.s, r.e]$∩$[q.s, q.e]$≠∅ holds for all r∈R with $r.e$≥t and $r.s$<t. All such rs can be found efficient by scanning a few of $List[i]$s for i∈A_4.

A_4 is scanned in a skipping manner. Specifically, $List[i]$ is scanned first for the biggest $i \in A_4$ which can be obtained from the smallest element in A_3. After this, apply $i' = oiPrev(i)$ iteratively to determine the next list $List[i']$ to be scanned till i'=0. According to Def. 1, most $List[i]$s ($i \in A_4$) are skipped over directly.

When $List[i]$ is scanned for $i \in A_4$, *oiSuc* is used again to avoid producing repeated results by finding out which part of the list should be scanned according to $j = oisuc(i) > i_0$ or not. These two subcases are illustrated in Fig. 4. In both cases, $[r.s, r.e] \cap [q.s, q.e] \neq \emptyset$ for $\forall r \in List[i]$ with $r.e \geq t$.



(a) Repeated but not output (b) Not Repeated partially

Fig. 4. Two case for $i \in A_4$

Case 4a: j=$oisuc(i)$>i_0 (Fig. 4(a)). In this case, $\langle r, q\rangle$ is not output before $List[i]$ is scanned. Thus, $List[i]$ is scanned backwardly till $r < t$ holds. For each scanned r, $\langle r, q\rangle$ is output.

Case 4b: j=$oisuc(i)$≤i_0 (Fig. 4(b)). In this case, $\langle r, q\rangle$ with $r.e \geq I.EA[j] \wedge r \in List[i]$ has been output when $List[j]$ is scanned, because such r is also indexed in $List[j]$. Thus, $List[i]$ is only scanned partially to find out all rs satisfying $t \leq r.e < I.EA[j]$, which can be accomplished via binary search to locate t in $List[i]$. For each scanned r, $\langle r, q\rangle$ is output.

Algorithm. Summarily, the ideas above is implemented as Alg. 3. It first computes i_0 (Line 1) and scans $List[i_0]$ thoroughly(Line 2). Then, it scans $List[i]$ for $i \in A_3$ (Line 3-Line 10). For each $i \in A_3$ (checked in Line 4), it accesses $I.SUC$ to obtain $oiSuc(i)$ (Line 5) and decides how to scan $List[i]$. Line 6-Line 7 is for *case 3a* and Line 8-Line 9 is for *case 3b*. At the end of A_3, point t and the biggest $i \in A_4$ is obtained (Line 11). Then, the algorithm scans $List[i]$ for $i \in A_4$ skippingly(Line 12-Line 18). For each $i \in A_4$ (checked in Line 12), it accesses $I.SUC$ to obtain $oiSuc(i)$ (Line 13) and decides how to scan $List[i]$. Line 14-Line 15 is for *case 4a* and Line 16-Line 17 is for *case 4b*. After that, it applies $oiPrev$ repeatedly to process next list till $i = 0$(Line 18).

Algorithm 3: OverlapIntervalSearch (q, I)

Input: $\langle , q, q.T\rangle$, index I of relation R
Output: $\{\langle q, r\rangle | r \in R \wedge r.T \cap q.T \neq \emptyset\}$

```
1  i0 ← BinarySearcch(qe, I.EA);
2  output⟨q, r⟩ for ∀r∈ I.List[i0] s.t. r.s≤ q.e;
3  i ← i0 − 1;
4  while I.EA[i] ≥ qs do
5      j ← I.SUC[i];
6      if j > i0 then                          // 3a
7          output⟨q, r⟩ for ∀r ∈ I.List[i];
8      else                                    // 3b
9          output⟨q, r⟩ for ∀r ∈ I.List[i] s.t.
           I.EA[i] ≤ r.e < L.EA[j];
10     i ← i − 1;
11 t ← I.EA[i + 1];
12 while i > 0 do
13     j ← I.SUC[i];
14     if j > i0 then                          // 4a
15         output⟨q, r⟩ for ∀r ∈ I.List[i] s.t.
           r.e ≥ t;
16     else                                    // 4b
17         output⟨q, r⟩ for ∀r ∈ I.List[i] s.t.
           t ≤ q.e < L.EA[j];
18     i ← oiPrev(i);
```

Example 5: Given tuple $\langle q, [10, 11]\rangle$ and the index in Fig. 2(c). Alg. 3 runs as follows. First, q is aligned to the endpoint array and four subsets are obtained, *i.e.,* A_1=$\emptyset$, A_2=$\{7\}$, A_3=$\{6\}$ and A_4=$\{0, 1, 2, 3, 4, 5\}$. Then, $List[7]$ is scanned and $\langle q, r_6\rangle$ is output. Then, $List[6]$ is scanned according to *case 3a* and $\langle q, r_{10}\rangle$, $\langle q, r_{11}\rangle$ and $\langle q, r_5\rangle$ are output. After that, t=10 and 5=$\max_{i\in A_4} i$. Thus, $List[5]$ is scanned partially according to *case 4b* but nothing is output. Then, since $oiPrev(5) = 4$, $List[4]$ is scanned backwardly till $r.e < 10$ according to *case 4a*, and $\langle q, r_{14}\rangle$,$\langle q, r_{13}\rangle$ and $\langle q, r_7\rangle$ are output. Finally, since $oiPrev(4) = 0$, the algorithm terminates. □

Analysis. In what follow, we first prove the correctness of Alg. 3, and then analyse its complexity.

For correctness, we show that the output of Alg. 3 are join results and each result is output once exactly. The former statement is straightforward. In fact, the algorithm produces result only in Line 2, Line 7, Line 9, Line 15, and Line 17 and the overlapping conditions there are easy to check. For the latter, we need the lemma below.

Lemma 8: *Given tuple* $\langle q, [q.s, q.e]\rangle$ *and the* O2i *index* I *of a relation* R*, if* i *satisfies* $I.EA[i\text{-}1] < q.e \leq I.EA[i]$ *and* $[r.s, r.e] \cap [q.s, q.e] \neq \emptyset$ *for a tuple* $r \in R$*, then there is* j *such that* $j \leq i$ *and* $r \in I.List[j]$.

Proof: Since $[r.s, r.e] \cap [q.s, q.e] \neq \emptyset$, $r.s \leq q.e$ holds.

If $r.s = q.e$, we have $I.EA[i\text{-}1] < r.s \leq I.EA[i]$. According to Line 13-Line 17 of Alg. 2, $r \in I.List[i]$.

Otherwise, $r.s < q.e$. Since $I.EA$ records end points of all intervals in R, there is a unique $j(\leq i)$ such that $I.EA[j\text{-}1] < r.s \leq I.EA[j]$. According to Line 13-Line 17 of Alg. 2, r is in $I.List[j]$. □

Lemma 9: *Under the conditions of Lemma 8,* $\langle q, r\rangle$ *is output by Alg 3 precisely once.*

Proof: Let $R.e = I.EA[n']$. Generally, we can assume $q.e \leq R.e$. In fact, if $q.e > R.e$, then $[q.s, q.e] \cap [r.s, r.e] = [q.s, R, e] \cap [r.s, r.e]$.

Since $I.EA$ records end points of all intervals in R, there is a unique i_0 satisfying $I.EA[i_0 - 1] < q.e \leq I.EA[i_0]$. i_0 is obtained in Line 1 of Alg. 3. According to Lemma 8, there is a largest integer i such that $i \leq i_0$ and $r \in List[i]$. The choice of i means that $I.SUC[i] > i_0$. Next, we show that $List[i]$ will be scanned by Alg. 3, and $\langle q, r\rangle$ is output then but is never output again elsewhere.

If $r.s = r.e$, then conclusion is obvious. In fact, according to Alg. 2, r is only indexed in $List[i]$. Furthermore, since $r.e = I.EA[i] \geq q.s$ and $i \leq i_0$, $List[i]$ is scanned during Line 4 -Line 10 of Alg. 3 being executed. Thus, $\langle q, r\rangle$ is output.

Otherwise, $r.s < r.e$. According to Line 13-Line 17 of Alg. 2, r may be indexed in other list $I.List[i']$. If so, we know $i' < i$ and $i = oiSuc^{(a)}(i', n')$ for some $a > 0$, because Alg. 2 only indexes r along an $oiCHAIN$ (see Alg. 2). To complete the proof, we show that Alg. 3 outputs $\langle q, r\rangle$ when $List[i]$ is scanned and is not output again when $I.List[i']$ is scanned. We discuss two cases.

Case 1: $I.EA[i] \geq q.s$. $List[i]$ is scanned during Line4-Line10 of Alg. 3 being executed. Now, we consider what happens when $I.List[i']$ is scanned. Since $oiSuc(i', n') = I.SUC[i'] \leq i$, Alg. 3 executes Line 9 or Line 17. Since $r.e \geq I.EA[i] > I.EA[I.SUC[i']]$, $\langle q, r\rangle$ will not be output again.

Case 2: $I.EA[i] < q.s$. We only need to show that $List[i]$ is scanned during Line 12-Line 18 of Alg. 3 being executed. If so, following the same discussion as *case 1*, we know $\langle q, r\rangle$ is output by

Alg 3 exactly once. For convenience, let v be the value of loop variable when Alg. 3 reaches Line 11.

First of all, we claim that the lowest bit of $(i)_2$ is a 0-bit. Otherwise, $I.SUC[i] = i + 1$ (Cor. 4). And then, $I.SUC[i] \leq i_0$ which conflicts with $I.SUC[i] > i_0$.

Now that the lowest bit of $(i)_2$ is a 0-bit, there exists some u such that $i = oiSuc(u, n')$. In other words, $oiSuc(u, n') = i < v < oiSuc(i, n')$. According to Cor. 5, there exits $k > 0$ such that $i = oiPrev^{(k)}(v)$, *i.e.,* $List[i]$ is scanned in during Line 12-Line 18 of Alg. 3 being executed. □

Now, we consider the complexity of Alg. 3. Let n_z be the size of output. Then, Line 1 takes $O(\log n)$ time. Line 2 can be implemented in $O(n_z)$ time (see Remark in Sec. 5). The while-loop from Line 4 to Line 10 takes $O(n_z)$ time, since both Line 7 and Line 9 can visit $List[i]$ forwardly. Finally, the while-loop from Line 12 to Line 18 takes $O(n_z + \log^2 n)$ time. In fact, this loop visits at most $\log n$ inverted lists (see Line 18 and Def. 1). Line 15 visits inverted lists backwardly each time. Totally, all executions of Line 15 take $O(n_z + \log n)$ time. However, Line 17 takes $O(\log n)$ time to locate t each time when it visits $List[i]$. Therefore, all executions of Line 17 take totally$O(n_z + \log^2 n)$ time. Thus, Alg. 3 runs in $O(n_z + \log^2 n)$ time.

Summarily, we obtain the following theorem.

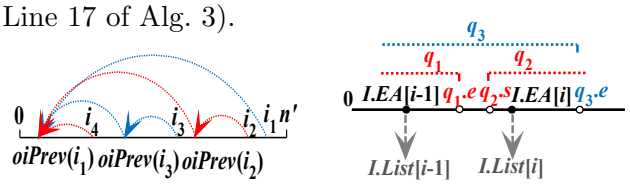
Theorem 10: *Via the* O2i *index of an n-tuple relation R, Alg. 3 evaluates the overlap join between a tuple q and R in* $O(n_z + \log^2 n)$ *time.*

6.2 Join two relations

Now, we consider how to evaluate the overlap join over two relations R and R'. A straightforward method works as follows: build an O2i index I for R first and then invoke Alg. 3 to join each $q \in R'$ with R. According to Theorem 10, this method outputs all result in $O(n_z + m \log^2 n)$ time. However, when I can't be fit into main memory, this method incurs its high I/O costs which can be avoided by utilizing the nesting structures approximated by the O2i index.

The main idea comes from the nesting structures of $oiPrev$(see Lemma 1), which means that all lists visited in the skipping manner have the First-in-Last-out property (see Fig. 5(a)). This makes it possible to process all inverted lists $List[i]$ in decreasing order of i by delaying the join between tuples r and $List[oiPrev(i)]$ till $List[oiPrev(i)]$ is scanned. This can be accomplished by manage all such tuples via a stack S. Besides this, a linear list L is used to manage all tuples which want inverted lists being processed sequentially. In what follows, we introduce the maintenance of these data structures first, and then present the algorithm and optimize it further.

Maintenance of the stack. When tuple q is joined with $List[i]$, if q need be joined with $List[oiPrev(i)]$ later, then q is pushed into stack S. After all $List[j]$ $(oiPrev(i) < j < i)$ are processed, q is popped out to join with $List[oiPrev(i)]$. All information about this delayed join operation are record in S. Specifically, each element in S is a quadruple $\langle q, t_q, j_q, k_q \rangle$, which records how to join q with $List[j_q]$. If $j = I.SUC[j_q] > k_q$, then q is joined with $r \in List[j_q]$ which satisfies $r.e \geq t_q$(see Line 15 of Alg. 3). Otherwise, q is joined with $r \in List[j_q]$ which satisfies $I.EA[j] > r.e \geq t_q$(see Line 17 of Alg. 3).



(a) First-in-Last-out (b) From i to $i-1$

Fig. 5. Illustration of the main ideas in O2iJoin

Maintenance of the linear list. The linear list L is used to manage tuples $q \in R'$ which require inverted lists are processed in a non-skipping manner. Each unit in L is a pair $\langle q, k_q \rangle$, which describes how to join q with $List[i]$ when $List[i]$ is processed. If $j = I.SUC[i] > k_q$, then q is joined with all $r \in I.List[i]$ (see Line 7 of Alg. 3). Otherwise, q is joined with $r \in I.List[i]$ which satisfies $r.e < I.EA[j]$ (see Line 9 of Alg. 3).

The maintenance of L is illustrated in Fig. 5(b), where $I.EA$ is used to determine tuples to be removed from L and tuples to be added into L. For the former case, when q with $\langle q, k_q \rangle \in L$ is joined with $List[i]$ and $I.EA[i] \geq q.s > I.EA[i-1]$, then $\langle q, k_q \rangle$ is removed from L and $\langle q, I.EA[i], i-1, k_q \rangle$ is pushed into S. For the latter case, after each q with $\langle q, k_q \rangle \in L$ is joined with $List[i]$, for these unprocessed $q \in R'$ satisfying $I.EA[i] \geq q.e > I.EA[i-1]$, q is joined with $List[i]$ and then $\langle q, i-1 \rangle$ is added into L.

Algorithm 4: o2iJoin (R, R')

Input: two temporal relations R and R'
Output: $\{\langle q, r\rangle | r{\in}R,\ q{\in}R',\ r.T \cap s.T \neq \emptyset\}$

```
1  if |R|>|R'| then switch the roles of R,R';
2  I_R ← createO2iIndex(R);
3  Sort R' such that q_1.e ≥ q_2.e ≥ ... ≥ q_m.e;
4  j ← 1,   i ← |I.EA|,   S ← ∅
5  while q_j.e ≥ I.EA[i] do
6      if q_j.s ≤ I.EA[i] then add ⟨q_j, i⟩ to L;
7      j ← j + 1;
8  while i > 0 and j ≤ m do
9      S.pop(⟨q, t_q, j_q, k_q⟩) till j_q < i;
10     foreach popped ⟨q, t_q, j_q, k_q⟩ do
11         if I.SUC[i] > k_q then
12             output ⟨r, q⟩ for ∀r ∈ I.List[i]
               s.t. r.e ≥ t_q;
13         else
14             output ⟨r, q⟩ for ∀r ∈ I.List[i]
               s.t. I.EA[k_q] > r.e ≥ t_q;
15             S.push(⟨q, t_q, oiPrev(j_q), k_q⟩);
16     foreach ⟨q, k_q⟩ ∈ L do
17         if I.SUC[i] > k_q then
18             output ⟨r, q⟩ for ∀r ∈ I.List[i];
19         else
20             i' ← I.SUC[i];
21             output ⟨r, q⟩ for ∀r ∈ I.List[i]
               s.t. I.EA[i] ≤ r.e < I.EA[i'];
22         if q.s > I.EA[i − 1] then
23             delete ⟨q, k_q⟩ from L;
24             S.push(⟨q, I.EA[i−1], i−1, k_q⟩);
25     while q_j.e > I.EA[i − 1] do
26         output ⟨r, q_j⟩ for ∀r ∈ I.List[i] s.t.
           r.s < q_j.e;
27         Insert ⟨q_j, i − 1⟩ into L;
28         j ← j + 1;
29     i ← i − 1;
```

Algorithm. The ideas above are implemented as the O2iJoin algorithm (see Alg. 4). It first builds O2i index I over smaller relation R (Line 1-Line 2). Then, it sorts R' in descending order of $r.e$ (Line 3) and does initialization (Line 4). Line 5-Line 7 initializes L to contain all tuples $q \in R'$ with $q.e \geq I.EA[n']$ and $q.s \leq I.EA[n']$. Notice that, all skipped tuples is not overlapping any intervals in R. Line 8-Line 29 process $List[i]$ in three stages. The first stage joins $List[i]$ with some tuples q on the top of S and updates the information about these tuples (Line 9-Line 15). Line 9 pops from S all quadruple $\langle q, t_q, j_q, k_q\rangle$ satisfying $j_q = i$. For each popped $\langle q, t_q, j_q, k_q\rangle$, q is joined with different parts of $List[i]$ according to $I.SUC[i] > k_q$ or not (Line 12, Line 14). Then, the quadruple is updated as $\langle q, t_q, oiPrev(j_q), k_q\rangle$ and pushed into S again (Line 15). The second stage joins $List[i]$ with each tuple in L and moves some tuples from L to S if necessary (Line 16-Line 24). For each $\langle q, k_q\rangle \in L$, it first joins q with different parts of $List[i]$ according to $I.SUC[i] > k_q$ or not (Line 18 and Line 21) and then check whether $q.s > I.EA[i-1]$ or not(Line 22). If so, $\langle q, k_q\rangle$ is removed from L (Line 23) and $\langle q, I.EA[i], i-1, k_q\rangle$ is pushed into S (Line 24). The third stage introduces new tuples into L (Line 25-Line28). Notice that, j traces next unprocessed tuple in R'. If $q_j.e > I.EA[i-1]$, then q is joined with $List[i]$ (Line 26) and $\langle q_j, i-1\rangle$ is added into L (Line 27). After these stages, the procession of $List[i]$ is completed. The algorithm turns to process $List[i-1]$ repeatedly till all inverted lists are processed.

The correctness of Alg. 4 follows from the correctness of Alg. 3. Moreover, the time complexity of Alg. 4 is $O(n_z + m\log^2 n)$, according to Theorem 10, because Alg. 4 just finishes all work of Alg. 3 for all tuples of R' concurrently.

Optimization. The stack S in Alg. 4 may require a huge space. In fact, S manages almost all tuples q of R', because each time when $\langle q, t_q, j_q, k_q\rangle$ is popped from S, $\langle q, t_q, oiPrev(j_q), k_q\rangle$ will be pushed into S again till $oiPrev(j_q)$=0. However, the lemma below tell us that all different values of j_qs in S forms a small set, which means there are always only a few of inverted lists waiting to be processed skippingly. This suggests us a method to reduce the space costs by loading these lists into memory and join with tuples eagerly.

Lemma 11: *During the running of Alg. 4, it is always true that* $|\{j_q | \langle q, t_q, j_q, k_q\rangle \in S\}| \leq \log n$.

Proof: Consider the time when $List[i]$ is processing. Assume $J_S = \{j_q | \langle q, t_q, j_q, k_q\rangle \in S\}$ contain z integers $j^{(1)}{<}j^{(2)}{<}\cdots{<}j^{(z)}$. For each $j^{(l)} \in J_S$, define $i^{(l)}$ to be the minimum i' such that $\langle q, t_q, j^{(l)}, k_q\rangle$ is pushed into S when $List[i']$ is processed. Notice that, $j^{(l)} = oiPrev(i^{(l)})$, $i \leq i^{(l)}$ and $i \geq j^{(l)}$. Moreover, Lemma 1 tells us that

$j^{(1)}<j^{(2)}<\cdots<j^{(z)}\leq i\leq i^{(z)}<\cdots<i^{(2)}<i^{(1)}$.

We estimate z by considering $(j^{(l)})_2$ and $(j^{(l+1)})_2$. Assume $(i^{(l)})_2= b_1\cdots b_{k-1}10\cdots 0$, *i.e.,* the lowest 1-bit is the kth bit. We have $(j^{(l)})_2 = b_1\cdots b_{k-1}00\cdots 0$, addcording to Def. 1. Since $j^{(l)}<j^{(l+1)}<i^{(l+1)}<i^{(l)}$, it must be the case that $(i^{(l+1)})_2=b_1\cdots b_{k-1}0b_{k+1}\cdots b_M$, where $M=\lceil\log n\rceil$. Moreover, there exist at least two 1-bits in $b_{k+1}\cdots b_M$. Otherwise, we will have $j^{(l)}=j^{(l+1)}$. Therefore, $(j^{(l+1)})_2= b_1\cdots b_{k-1}0c_{k+1}\cdots c_M$ and there exists at least one 1-bit in $c_{k+1}\cdots c_M$. Thus, $(j^{(l+1)})_2$ can be obtained from $(j^{(l)})_2$ by changing at least one of the lowest continuous 0-bits into 1-bits.

Since $j^{(1)}\neq 0$, $(j^{(1)})_2$ has at most $\log n$ contiguous 0-bits in its lowest positions, $z\leq\log n$. □

Lemma 11 means that if a fixed-size buffer is assigned for S, then each time when the buffer is full, it can be emptied by loading at most $\log n$ lists and joining S with these lists. Even, S can be replaced with the eager-skip strategy below. O2iJoin uses a buffer to cache $\log n+1$ lists, including the list scanned currently and the $\log n$ lists for emptying S. Each time when a tuple need be pushed into S, it is joined in a eager-skip manner with the $\log n$ lists cached in the buffer. To implement this strategy, O2iJoin must be modified as follows.

- In Line 2, after index I is built, obtain the maximum length l_{max} of all inverted lists.
- In Line 4, besides these initialization, malloc a buffer B of size $O(l_{max}(\log n+1))$.
- Replace Line 9-Line 15 with a single line, *i.e.,* "load $I.List[i]$ into B if it is not there".
- Replace Line 24 with the following lines.
  24.1 $i'\leftarrow i-1$;
  24.2 **while**$(i'>0)$ **do**
  24.3 load $I.List[i']$ into B if it is not there;
  24.4 **if**$(I.SUC[i']>k_q)$**do**
  24.5 do Line 12 for $List[i']$;
  24.6 **else** do Line 14 for $List[i']$;
  24.7 $i'\leftarrow oiPrev(i')$
- In Line 29, free $I.List[i]$ from B;

Now, we get the theorem below.

Theorem 12: *Given temporal relations R and R' with $|R|=n$ and $|R'|=m$. Let l_{max} be the maximum length of all inverted lists of R's* O2i *index I. The* O2iJoin *algorithm evaluates overlap join between R and R' in $O(n_z+m\log^2 n)$ CPU time by using a buffer of size $O(l_{max}(\log n+1))$ and reading each inverted list of I only once.*

Example 6: Consider running of Alg. 4 on R in Fig. 1(a) and $R'=\{\langle q_0,[13,15]\rangle,\langle q_1,[11,13]\rangle, \langle q_2,[7,10]\rangle\}$. R's index I is in Fig. 2(c). The buffer B can accommodate $\log n'+1=3$ inverted lists. After initialization, $L=\{\langle q_1,7\rangle\}$ and q_0 is ignored.

Then, $I.List[7]$ is loaded into B to join with q_1. Line 18 outputs $\langle r_6,q_1\rangle$. Since $q_1.s>I.EA[6]$, q_1 is removed from L. Then, since $oiPrev(7)=6$, $I.List[6]$ is loaded into B to join with q_1. Line 24.5 outputs $\langle r_{10},q_1\rangle$, $\langle r_{11},q_1\rangle$ and $\langle r_5,q_1\rangle$. Furthermore, since $oiPrev(6)=4$, $I.List[4]$ is loaded into B to join with q_1. Line 24.6 outputs $\langle r_{13},q_1\rangle$. After that, the algorithm frees $List[7]$ from B and turns to process $List[6]$.

Now, $B=\{List[6],list[4]\}$ and $L=\emptyset$. Line 9-Line 24 do nothing. Since $q_2.e\geq I.EA[6]$, Line 26 outputs $\langle r_{10},q_2\rangle$,$\langle r_{11},q_2\rangle$ and $\langle r_5,q_2\rangle$. Then, Line 27 inserts $\langle q_2,6\rangle$ into L. After that, the algorithm frees $List[6]$ from B and turns to process $List[5]$.

Then, $B=\{list[4]\}$, $L=\{\langle q_2,6\rangle\}$. Therefore, $I.List[5]$ is loaded into B to join with q_2. Line 21 outputs $\langle r_4,q_2\rangle$. Then, since $q_2.s>I.EA[4]$, q_2 is removed from L. Notice that, $oiPrev(5)=4$ and $I.List[4]$ is in B. Thus, q_2 is joined with $List[4]$ directly. Line 24.5 outputs $\langle r_7,q_2\rangle$,$\langle r_{13},q_2\rangle$. After that, the algorithm frees $List[5]$ from B and turns to process $List[4]$.

Since $j=|R'|$, the algorithm terminates. □

Remark. Theorem 12 holds only if all tuples of R' can be accommodated in L. Otherwise, the O2i index I of R must be loaded into memory more than once. In fact, if L can only cache m' tuples of R', then the join operation can be accomplished by running Alg. 4 $\lceil m/m'\rceil$ passes. In each pass, m' tuples of R' are loaded into L and joined with R. As a whole, each inverted list of I needs to be loaded into memory no more than $\lceil m/m'\rceil$ times.

7 Experimental Evaluation

This section compares O2iJoin with the state-of-the-art methods empirically via two sets of experiments. The first one compares their performance on three real datasets, and the second one uses synthetic data to compare the impacts of long-

lived tuples on performance of different algorithms.

Setup. We compare O2iJoin with four algorithms, *i.e.,*DIP [1], EBI [3], FS [2] and OIP [4]. DIP and OIP are the state-of-art partition-based algorithms. EBI and FS are the state-of-the-art index-based algorithms. Authors of [1, 3] kindly provide us their source code. All algorithms are implemented in C++ and compiled with o3 flag on a 2x Intel(R) Core(TM) i3-2120@3.30GHz machine with 32GB main memory running a Linux 6.4 operating system. The O2i index is written into disks with block size of 10K bytes. Other algorithms run in memory without optimizations of caching or multicore. We compare their total running time, which including costs for data loading, data partitioning, data indexing and overlap joining.

EXP1: Real datasets. We use three real datasets, *i.e.,* GREED [1],Webkit [30] and Incumbent [31]. GREEND[1], which contains very short intervals, is public and contains detailed power usage information obtained through a measurement campaign in households in Austria and Italy from January 2010 to July 2017. The Incumbent contains 300,000 tuples, which record the history of employees assigned to projects over a 17-year period at a granularity of days. The Webkit contains 1,540,000 tuples, which record the history of files of the svn repository of the webkit project over a 11-year period at a granularity of milliseconds. The valid times indicate the periods when a file did not change. Both Webkit and Incumbent contains many long-lived intervals.

For each dataset, let R' be the whole dataset and R be a subset to include 25%,50%,75% and 100% tuples of the whole dataset respectively. Five algorithms evaluate overlap join between R and R', where R is the indexed in O2iJoin. The results are reported in Fig. 6.

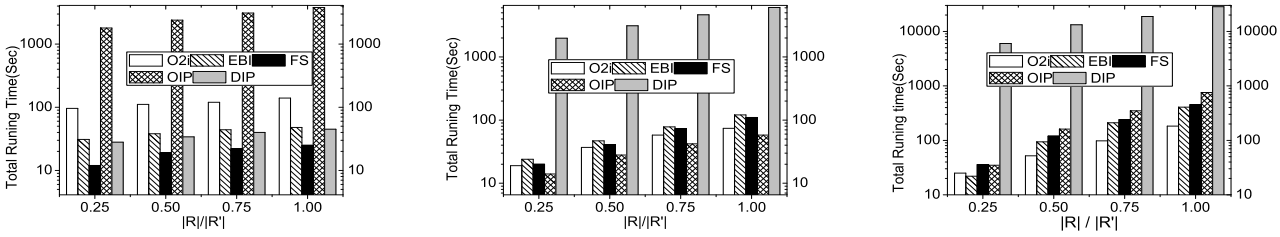
As expected, the execution time of all methods rises as we increase the ratio $|R|/R'$. Both DIP and OIP are dataset sensitive. Their performances vary widely between short-lived intervals(*i.e.,* GREEN(Fig 6(a))) and long-lived intervals(*i.e.,* Webkit(Fig. 6(c)) and Incumbent(Fig. 6(b))). DIP prefers short intervals while OIP prefers long intervals. O2iJoin runs slower than DIP,EBI,FS on short intervals, and faster on long intervals. This is because the nesting structures among long intervals are more obvious than those among short intervals and the O2iJoin can utilize such nesting structures approximated by the O2i index efficiently.

EXP2: Synthetic datasets. This experiment compares O2iJoin with other four algorithms by varying the number of long-lived intervals in 15 synthetic datasets. Each dataset contains 1M tuples spanning range $[0, 2^{38}]$. Short intervals have a maximum duration of 0.01% of this range. Long intervals have a duration up to 8% and an average duration of 4%. The lengths of short intervals distribute uniformly. While the lengths of long intervals follows uniform distribution, or Gaussian distribution, or Zipf distribution. For each kinds of distribution, the percentage of long intervals are set to be 0%, 5%,10%,20% and 40% respectively. Algorithms run to join each dataset with itself. Experimental results are presented in Fig. 7.

We find that (1) The running time of each algorithm rises when the number of long intervals increases. And, the effects of different distributions are not obvious. This is because the number of long-lived tuples affect all algorithms by changing their indices, (number of) data partitions and joining procedures. (2)Partition-based algorithms perform poor on these synthetic datasets. For example, DIP always costs over 10,000 seconds and OIP always costs near 1000 seconds. However, O2iJoin, EBI, and FS only cost a few hundred seconds. The inefficiency of partition-based algorithms stems from the fact that they must generate many data partitions and then join them one by one. (3) O2iJoin, EBI, and FS are all efficient on these dataset. This is because all indices adopted in these algorithms are flat structures which can support overlap join with low CPU costs.

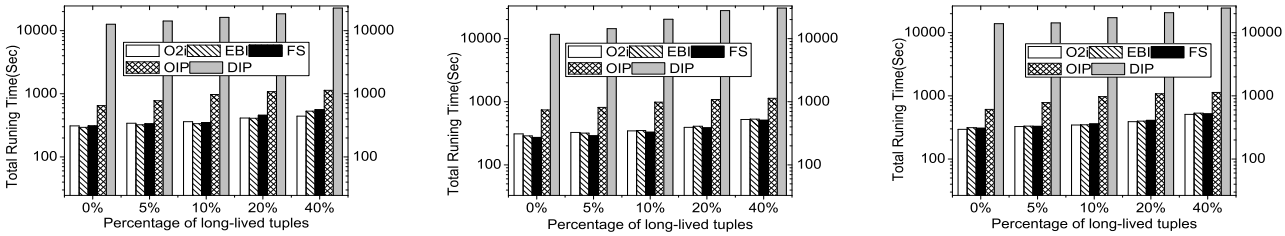
8 Conclusion and future work

This paper proposes an algorithm named O2iJoin to evaluate overlap join over temporal relations via a novel two-layer index named O2i index. The index can be built on-the-fly in $O(n \log n)$ time and $O(n \log n)$ space. The O2iJoin algorithm loads the index only once and evaluates overlap join in $O(n_z + m \log^2 n)$ CPU time. Experiments on real datasets and synthetic datasets show that this new join algorithm is as competitive as the state-of-the-



(a) GREEND (b) Incumbent (c) Webkit

Fig. 6. Performance on real datasets



(a) Uniform (1M tuples) (b) Gaussian (1M tuples) (c) Zipf (1M tuples)

Fig. 7. Performance on synthetic datasets

art algorithms.

Two interesting directions for the future work are to: (1)optimize O2iJoin further with caching or parallel mechanism provided by the hardware;(2)incrementally update the O2i index such that it can be used to support other queries.

References

[1] F. Cafagna and M. H. Böohlen. Disjoint interval partitioning. *VLDB Journal*. 26(3):447-466, 2017.

[2] Panagiotis Bouros, Nikos Mamoulis. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. the VLDB Endowment, 26(10): 1346-1347.

[3] D. Piatov, S. Helmer, A. Dignös. An Interval Join Optimized for Modern Hardware. In *Proc. the 32th Int'l Conf. Data Engineering (ICDE)*, May 2016, pp.1098-1109.

[4] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap Interval Partition Join. In *Proc. the ACM SIGMOD Conference*, June 2014, pp.1459 – 1470.

[5] D. Gao, C.S. Jensen, R.T. Snodgrass, M. Soo. Join Operations in Temporal Databases. *VLDB Journal*, 2005, 14(1): 2-29.

[6] J. Enderle, M. Hampel, T. Seidl. Joining Interval Data in Relational Databases. In *Proc. the ACM SIGMOD Conference*, June 2004, pp.683-694.

[7] V.J. Tsotras, A. Kumar. Temporal Database Biliography Update. *ACM SIGMOD Record*, 25(1):41-51, 1996.

[8] G. Ozsoyoglu, R.T. Snodgrass. Temporal and Real-time Databases: A survey. *IEEE Trans. on Knoledge and Data Engineering*,7(4):513-532, 1995.

[9] R. Elmasri, G.T.J. Wuu, Y.J. Kim. The Time Index: An Access Structure for Temporal Data. In *proc. the 16th Int'l Conf. Very Large Data Bases(VLDB)*, Aug 1990, pp.1-12.

[10] D. Son, R. Elmasri. Efficient Temporal Join Processing Using Time Index. In *Proc. the 8th Int'l Conf. Scientific and Statistical Data Base Management(SSDM)*, Aug 1996, pp.252-261.

[11] M.D. Soo, R.T. Snodgrass, C.S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proc. the 10th Int'l Conf. Data Engineering (ICDE)*, April 1994, pp.282-292.

[12] I. Sitzmann, P.J. Stuckey. Improving Temporal Joins Using Histograms. In *Proc. the 10th Int'l Conf Database and Expert Systems Applications(DEXA)*, Sep 2000, pp.488-498.

[13] H. Lu, B.C. Ooi, K.L. Tan. On Spatially Partitioned Temporal Join. In *proc. the 20th Int'l Conf. Very Large Data Bases(VLDB)*, Aug 1994, pp.546-557.

[14] H. Shen, B.C. Ooi, H. Lu. The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. In *Proc. the 10th Int'l Conf. Data Engineering (ICDE)*, April 1994, pp.274-281.

[15] L. Arge, O. Procopiuc, S. Ramaswamy, et al. Scalable Sweeping-Based Spatial Join. In *proc. the 24th Int'l Conf. Very Large Data Bases(VLDB)*, Aug 1998, pp.570-581.

[16] M.L. Lo, C.V. Ravishankar. Spatial Hash-Joins. In *Proc. the ACM SIGMOD Conference*, June 1996, pp.247-258.

[17] J.M. Patel, D.J. Dewitt. Partition Based Spatial-Merge Join. In *Proc. the ACM SIGMOD Conference*, June 1996, pp.259-270.

[18] N. Koudas, K.C. Sevcik. Size Separation Spatial Join. In *Proc. the ACM SIGMOD Conference*, June 1997, pp.324-335.

[19] H. Gunadhi, A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proc. the 7th Int'l*

Conf. Data Engineering (ICDE), April 1991, pp.336-344.

[20] S.P. Rana, F. Fotouhi. Efficient Processing of Time-Joins in Temporal Data Bases. In *Proc. the 3th Int'l Conf. Database Systems for Advanced Applications(DASFAA)*, April 1993, pp.427-432.

[21] T.Y.C. Leung, R.R. Muntz. Stream Processing: Temporal Query Processing and optimization. In *A.U. Tansel et al(Eds.): Temporal Databases: Theory, Design, and Implementation*,1993, pp. 355-429.

[22] D. Pfser, S.J. Jensen. Incremental Join of Time-Oriented Data. In *Proc. the 8th Int'l Conf. Scientific and Statistical Data Base Management(SSDM)*, Aug 1996, pp.232-243.

[23] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, G.M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. the ACM SIGMOD Conference*, June 2001, pp.425-436.

[24] D. Zhang, V.J. Tsotras, B. Seeger. Efficient Temporal Join Processing Using Indices. In *Proc. the 18th Int'l Conf. Data Engineering (ICDE)*, April 2002, pp.103-113.

[25] D. Zhang, V.J. Tsotras. Index Based Processing of Semi- Restrictive Temporal Joins. In *the 9th Int'l Symp. Temporal Representation and Reasoning (TIME)*, July 2002, pp.70-77.

[26] H.P. Kriegel, M. Pötke, T. Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *Proc. the 26th Int'l Conf. Very Large Data Bases(VLDB)*, Aug 2000, pp.407-418.

[27] H. Samet. Foundation of Multidimensional and metric data structures. Morgan Kaufmann Publishers Inc., San Fransisco, CA, USA, 2005.

[28] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger. The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. the ACM SIGMOD Conference*, June 1990, pp.322-331.

[29] N. Beckmann, B. Seeger. A revised R^*-tree in comparison with related index structures. In *Proc. the ACM SIGMOD Conference*, June 2009, pp.799-812.

[30] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, A. Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *Proc. the ACM SIGMOD Conference*, June 2013, pp.701-712.

[31] V.J. Tsotras, N. Kangelaris. The Snapshot Index: An I/O Optimal access method for timeslice queries. *Infomation System*, 1995, 30(3):237-260.

[32] The webkit open source project. *http://www.webkit.org*, 2015.

[33] J. Gendrano, R. Shah, R. Snodgrass, J. Yang. University information system (uis) dataset. TimeCenter CD-1, 1998.