Overlap Interval Partition Join

Anton Dignös
Department of Computer Science
University of Zürich, Switzerland
adignoes@ifi.uzh.ch

Michael H. Böhlen
Department of Computer Science
University of Zürich, Switzerland
boehlen@ifi.uzh.ch

Johann Gamper
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
gamper@inf.unibz.it

ABSTRACT

Each tuple in a valid-time relation includes an interval attribute T that represents the tuple's valid time. The overlap join between two valid-time relations determines all pairs of tuples with overlapping intervals. Although overlap joins are common, existing partitioning and indexing schemes are inefficient if the data includes long-lived tuples or if intervals intersect partition boundaries.

We propose *Overlap Interval Partitioning* ($\mathcal{OIP}$), a new partitioning approach for data with an interval. $\mathcal{OIP}$ divides the time range of a relation into k base granules and defines overlapping partitions for sequences of contiguous granules. $\mathcal{OIP}$ is the first partitioning method for interval data that gives a *constant clustering guarantee*: the difference in duration between the interval of a tuple and the interval of its partition is independent of the duration of the tuple's interval. We offer a detailed analysis of the *average false hit ratio* and the *average number of partition accesses* for queries with overlap predicates, and we prove that the average false hit ratio is independent of the number of short- and long-lived tuples. To compute the overlap join, we propose the *Overlap Interval Partition Join* (OIPJOIN), which uses $\mathcal{OIP}$ to partition the input relations on-the-fly. Only the tuples from overlapping partitions have to be joined to compute the result. We analytically derive the optimal number of granules, k, for partitioning the two input relations, from the size of the data, the cost of CPU operations, and the cost of main memory or disk IOs. Our experiments confirm the analytical results and show that the OIPJOIN outperforms state-of-the-art techniques for the overlap join.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Query processing*

Keywords

Temporal databases; Overlap join; Interval partitioning

1. INTRODUCTION

A key operation in valid-time databases is the *overlap join* [11]: given two valid-time relations $\mathbf{r}$ and $\mathbf{s}$, find all pairs of tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$ with overlapping intervals, i.e., $r.T \cap s.T$. The overlap join gives the query optimizer an efficient option if other predicates are absent, exhibit a poor selectivity, or must be evaluated after the overlapping interval has been computed. For instance, to find employees who are employed during at least 5 months when a project is ongoing, we first must determine the overlapping interval between an employee and a project, and then check that the duration of the overlapping interval is at least 5 months. Our goal is an efficient join for interval data that offers the query optimizer a viable option when other joins do not perform well.

Partitioning techniques for interval data associate each partition with a *partition interval*. Each tuple is stored in the best fitting partition, i.e., the partition interval must cover the interval of the tuple, and there may not exist a smaller partition interval that covers the interval of the tuple. As an example, consider a partition p with partition interval $[2012\text{-}1, 2012\text{-}4]$ and a tuple s with interval $[2012\text{-}2, 2012\text{-}3]$. Tuple s can be stored in partition p since 2012-2 $\geq$ 2012-1 and 2012-3 $\leq$ 2012-4, and it is indeed stored in p if and only if there is no other partition with a smaller partition interval that covers the interval of s. Since a partition interval is usually larger than the intervals of the tuples in this partition, we get *false hits* when searching in a partition for tuples that overlap a query interval (a false hit is a tuple that is fetched with a partition but does not contribute to the result). False hits increase the number of IOs, since more data must be fetched, and the number of CPU operations, since false hits must be detected and discarded. In order to reduce the number of false hits, it is possible to create more partitions. Many partitions, however, increase the number of IOs since we get more partially filled blocks. This increases the number of CPU operations for *searching and navigating* in the access structure.

This paper proposes the OIPJOIN, together with *Overlap Interval Partitioning* ($\mathcal{OIP}$), to efficiently compute the overlap join. $\mathcal{OIP}$ partitions the time range of a relation uniformly at a granularity that is given by k temporally disjoint granules of duration d. We create partitions for all sequences of adjacent granules. This approach gives a *constant clustering guarantee*, i.e., the difference in duration between a tuple and its partition is less than $2d$, independent of the duration of the tuple. The access structure of $\mathcal{OIP}$, termed *lazy partition list*, omits empty partitions without sacrificing performance or functionality. The OIPJOIN is *self-adjusting*, i.e, it automatically determines the optimal number of granules, k, that minimizes the overhead costs of the OIPJOIN.

Example 1. Figure 1 illustrates $\mathcal{OIP}$ with $k = 4$ granules for two relations $\mathbf{r}$ and $\mathbf{s}$. The time range of $\mathbf{r}$ is $[2012\text{-}5, 2012\text{-}11]$, and the granules have a duration of $\lceil \frac{2012\text{-}11 - 2012\text{-}5 + 1}{4} \rceil = \lceil \frac{7}{4} \rceil = 2$ months. This is the granularity at which relation $\mathbf{r}$ is partitioned. The partitions span 2, 4, 6, or 8 months. Similarly, relation $\mathbf{s}$ is par-

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2612175.

titioned over its time range [2012-1, 2012-12]. The granules have a duration of 3 months, and the partitions span 3, 6, 9, or 12 months. For the OIPJOIN, $\mathbf{r} \bowtie_{\mathbf{r}.T \cap \mathbf{s}.T} \mathbf{s}$, we process for each partition in $\mathbf{r}$ the overlapping (relevant) partitions in $\mathbf{s}$. For instance, for the partition that contains r_1 and r_2, three partitions in $\mathbf{s}$ are processed, yielding three false hits, namely $\{s_6\}$ for r_1 and $\{s_3, s_5\}$ for r_2. For the partition that contains r_3, two partitions in $\mathbf{s}$ are processed, and there are no false hits. Overall, five partitions of $\mathbf{s}$ are accessed with three false hits and eight result tuples.

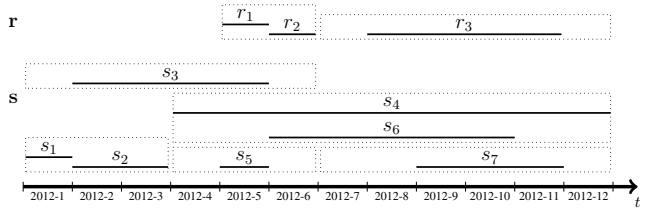


Figure 1: Sample Relations and $\mathcal{OIP}$.

False hits and partition accesses incur overhead costs for the OIPJOIN. The number of false hits and the number of partition accesses are inversely related. Increasing the number of granules k (i.e., shorter granules and more partitions) increases the number of partition accesses, but decreases the number of false hits, and vice versa. We analytically derive the k that minimizes the overhead costs by adapting to the size of the two relations, the cost of CPU operations, and the cost of IOs. Instead of assuming a dominating cost factor, we propose a cost model that accounts for CPU and IO costs. Note that IO costs can be memory IOs or disk IOs. A main memory IO is faster than a disk IO, but slower than a CPU operation [20]. Since data are transferred in chunks from the memory to the processor, it is favorable to store tuples in contiguous main memory blocks.

Summarizing, our technical contributions are as follows:

- We introduce $\mathcal{OIP}$ as partitioning strategy for the OIPJOIN. $\mathcal{OIP}$ offers a *constant clustering guarantee*, which ensures that the join does not deteriorate. The difference in duration between a tuple and its partition is less than two granules.
- We provide a detailed analysis of the *average false hit ratio* (AFR) and the *average number of partition accesses* (APA) for $\mathcal{OIP}$. We prove that AFR for uniformly distributed query intervals is smaller than $\frac{1}{k}$ and independent of the number of short- and long-lived tuples.
- The OIPJOIN is *self-adjusting*, i.e, it automatically determines the optimal number of granules k. We develop a cost function for the OIPJOIN and minimize this cost function to get the optimal number of granules k to partition the relations. k minimizes the overhead costs due to false hits and partition accesses for IO costs $c\_io \geq 0$ and CPU costs $c\_cpu \geq 0$.
- We describe an implementation of the OIPJOIN based on $\mathcal{OIP}$ and compare it with self-adjusting overlap joins based on quadtree, loose quadtree, segment tree, and relational interval tree. The experiments confirm that the OIPJOIN outperforms these approaches.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides preliminaries. Section 4 describes overlap interval partitioning ($\mathcal{OIP}$) and its implementation. Section 5 analytically investigates the average false hit ratio (AFR) and the average number of partition accesses (APA) of $\mathcal{OIP}$. Section 6 describes the OIPJOIN and derives the optimal value for k. Section 7 reports the results of our empirical evaluation.

2. RELATED WORK

We describe, in turn, related work on (a) self-adjusting approaches that, as the OIPJOIN, adapt to the data and do not require user-specified parameters; (b) parameter-guided approaches that can/must be tuned with application-specific parameters; and (c) disk-based approaches that introduced some of the key concepts used in later works.

Self-Adjusting Approaches. The quadtree [10, 18] recursively subdivides the space into cells and places objects in the smallest enclosing cell.[1] Since split boundaries are propagated down the tree and objects are not allowed to overlap boundaries, small objects that overlap split boundaries end up high in the tree. Therefore the quadtree does not have a clustering guarantee. For instance, time range $[1, 32]$ is recursively split into $[1, 16]$ and $[17, 32]$ and so on, and a tuple with interval $[16, 17]$ is placed in the root. This yields many false hits for overlap predicates since all tree nodes that overlap the query interval need to be scanned. The quadtree relies on a hierarchical tree structure, and in order to navigate to nodes at lower levels, all parent nodes must be stored, even if they are empty. To avoid many partially filled blocks, density based splitting is used, which propagates tuples down the tree only when blocks are full. This, however, increases the number of false hits.

The loose quadtree [18, 19] addresses the limitations of the quadtree for small objects. It permits at each level partially overlapping cells. The amount of overlapping is determined by a user-specified cell expansion factor, $p > 0$, where $p = 1$ is widely accepted as the best value [23, 18]. An expanded cell has width $(1 + p) \cdot w$, where w is the width of a quadtree cell. For instance, time range $[1, 32]$ is recursively split into $[1, 24]$ and $[9, 32]$ and so on. A tuple with interval $[16, 17]$ is placed in either $[14, 17]$ or $[16, 19]$, which are the expanded quadtree cells for $[15, 16]$ and $[17, 18]$. The join performance deteriorates for long-lived tuples since the time ranges grow with a factor of two, i.e., the number of partitions for long-lived tuples is much lower than for small tuples. The loose quadtree provides a clustering guarantee that is not constant. The guarantee depends on the duration of a tuple and is weaker for longer tuples. For instance, for $p = 1$ and a relation that spans 2000 days, a tuple of duration 80 days can be in a partition that spans 250 days, yielding a difference of 170 days between the tuple and the associated partition. A tuple that spans 282 days can even be in a partition of 1000 days, which is a difference of 718 days.

The relational interval tree [14] implements Edelsbrunner's interval tree on top of a relational DBMS. The approach uses two B+-tree indices to index intervals according to a key and start point and end point. A query interval is first transformed into a key point list and a key range list, which in a second step are joined with the B+-tree indices. For instance, given an indexed time range of $[1, 64]$ and a query interval $[5, 7]$, the key point list is $\{32, 16, 8\}$ and the key range list $\{[4, 4], [5, 7]\}$. These lists are joined with the help of the two B+-tree indices to get the final result. Vari-

[1] In order to manage intervals with 2D access structures we omit the second dimension, which reduces 2D points and 2D rectangles to, respectively, 1D points and intervals.

ous join techniques based on the relational interval tree have been proposed [9], such as an Index-Based Loop Join and several partition based joins (Up-Down, Down-Down, Up-Up depending on tree traversal). In all these join techniques, long-lived tuples lead to many CPU operations since a large number of nodes must be joined. To get a better IO performance for block storage, the data can be clustered according to an index on either the start or the end points. Nevertheless, long-lived tuples deteriorate the performance since for long-lived tuples the clustering of the two indices varies more than for short-lived tuples.

The segment tree [4, 5] is an indexing technique for intervals. It builds disjoint segments (intervals) at the leaf level, using all start and end points in a relation. Internal nodes merge all segments of their children nodes. A tuple is associated with all sub-tree roots whose segment is completely covered by the tuple's interval. The segment tree efficiently retrieves all tuples that include a given time point. In order to compute an overlap join, possibly empty parent nodes must be scanned, and duplicated tuples that are assigned to multiple nodes must be fetched (IO cost) and identified (CPU cost). This is particularly expensive for long-lived tuples. For instance, for a relation with three tuples r_1, r_2, and r_3 with intervals $[1, 5]$, $[3, 9]$, and $[8, 9]$, respectively, we have at the leaf level (level 2) the four segments $[1, 2]$, $[3, 5]$, $[6, 7]$, and $[8, 9]$. At level 1, we have the segments $[1, 5]$ and $[6, 9]$, and at level 0 (root) the segment $[1, 9]$. Tuple r_2 is stored twice, namely in $[3, 5]$ and $[6, 9]$, and it must be read twice for the query interval $[5, 6]$.

Parameter-Guided Approaches. In [16], a spatially partitioned temporal join is proposed, where interval data is mapped to points in a two dimensional grid. Partitions are regions in the plane. Two relations are joined by determining for each partition of the outer relation the relevant partitions of the inner relation. Two implementations are proposed, namely to store partitions physically on disk blocks or to use spatial indices to index the regions of partitions. While existing spatial indices can be reused, long-lived tuples substantially increase the number of index nodes to scan. The number of partitions must be specified by the application.

The snapshot index [22] is an access method for disk resident transaction-time databases. Intervals in transaction-time databases are clustered since database modifications occur in increasing time order. Blocks are distinguished by usefulness according to an application parameter a that indicates the number of false hits a block is allowed to generate. Long-lived tuples are artificially deleted and re-inserted using controlled splits. Splitting is not a general solution for valid-time databases since it changes the meaning of tuples [8, 7]. Parameter a must be chosen as a tradeoff between artificial splits and false hits.

In [17], an approach is proposed for data that is stored in main memory. It is similar to the spatial hash join [15] and uses an R-Tree to group tuples into minimum bounding rectangles (MBRs). The tuples of one relation are stored in the leaf nodes, and the tuples of the other relation in the lowest internal node, where more than one child's MBR overlaps the tuple. The join is performed by joining leaf with internal nodes. The approach aims to reduce the number of CPU comparisons and requires three parameters: tree fanout, number of partitions, and cells per dimension.

Disk-Based Approaches. The size separation spatial join [13] is a partitioning strategy for the overlap join of disk resident spatial data and is similar to the quadtree. Instead of using a tree structure, the levels of the tree are mapped to sorted files. The join is then performed by a synchronized scan of two sorted files that represent the partitioned relations. The approach reduces IO and provides a good filling of blocks, but, due to the recursive space division, small objects are not guaranteed to be stored at a low level. Thus, the size separation spatial join has no clustering guarantee and may produce many false hits.

The grace partition join [21] is used for valid-time natural joins of disk resident data, i.e., overlap joins with additional equality predicates. It samples the relations to determine the partitions for the tuple intervals. A tuple is stored in the last partition it overlaps. Partitions are joined from the last to the first partition. Long-lived tuples that overlap several partitions are migrated to the next partition during join processing. The approach is only efficient for few long-lived tuples, where the overhead of migration is low.

The R*-tree [2, 3] uses MBRs to group nearby objects and stores object IDs in the leaf nodes. The internal nodes build an index on the leaf nodes using MBRs. MBRs of both leaf and internal nodes might overlap. The tree is expensive to construct due to the propagation of MBRs. Long-lived tuples increase the MBRs and produce more false hits (page faults). For overlap joins, it is necessary to follow multiple paths in the R*-tree.

3. PRELIMINARIES

We assume a discrete time domain, Ω^T, consisting of a linearly ordered set of time points. An interval T is a contiguous set of time points and is represented as a pair $[T_S, T_E]$, where T_S is the inclusive start point and T_E the inclusive end point. We use the following operations on time points and intervals: $x \in T$ if time point x is contained in interval T, i.e., $T_S \leq x \leq T_E$; $Q \cap T$ if Q and T intersect, i.e., there exists a time point x such that $x \in Q \wedge x \in T$; $T \subseteq U$ if interval T is contained in interval U, i.e., $\forall x \in T \Rightarrow x \in U$; $T_E - T_S$ determines the difference in number of time points between T_S and T_E; $T_S + x$ shifts time point T_S by x time points to the right, i.e., $T_E - T_S = x \Rightarrow T_S + x = T_E$; and $|T| = (T_E - T_S) + 1$ is the duration (length) of interval T.

We use tuple timestamping and associate each tuple with a single interval that represents the tuple's valid time. A temporal relation schema is represented as $R = (A_1, \ldots A_m, T)$, where $A_1 \ldots A_m$ are attributes with domain Ω_i and T is the interval attribute over $\Omega^T \times \Omega^T$. A tuple r over schema R is a finite set that contains for every A_i a value $v_i \in \Omega_i$ and for T an interval $[T_S, T_E] \in \Omega^T \times \Omega^T$. A temporal relation $\mathbf{r}$ over schema R is a finite set of tuples over R. A valid-time relation $\mathbf{r}$ spans time range $U = [U_S, U_E]$ if U_S is the smallest start time point of any tuple in $\mathbf{r}$ and U_E the largest end time point of any tuple in $\mathbf{r}$. We write l for the duration of the longest tuple in a relation $\mathbf{r}$, and λ for the duration of the longest tuple as a fraction of the time range, i.e., $\lambda = l/|U|$. We use indices r and s to distinguish between the outer and inner relation in joins, e.g., n_r and n_s are, respectively, the cardinality of the outer and inner relation in $\mathbf{r} \bowtie \mathbf{s}$.

4. OVERLAP INTERVAL PARTITIONING

In this section, we first define Overlap Interval Partitioning ($\mathcal{OIP}$) and show how the relevant partitions, i.e., partitions that overlap a query interval, are calculated. Second, we establish the constant clustering guarantee. Third, we show how to manage physical partitions of $\mathcal{OIP}$ with a lazy partition list that omits unused partitions.

4.1 Definition

$\mathcal{OIP}$ divides a time range U into k equally sized granules of duration d, which define the base granularity of the partitioning (we discuss in Section 6.2 how to derive k).

Definition 1. (*$\mathcal{OIP}$ Configuration*) Let $\mathbf{r}$ be a temporal relation with time range $U = [U_S, U_E]$. An *$\mathcal{OIP}$ configuration* for a given k is a triple (k, d, o), where $d = \lceil \frac{|U|}{k} \rceil$ is the duration of each granule and $o = U_S$ is the start point of the partitioned time range.

A partition interval spans a sequence of one or more adjacent granules. A tuple is assigned to the smallest partition whose partition interval completely covers the tuple's interval.

Definition 2. (*$\mathcal{OIP}$ Partition*) Let $\mathbf{r}$ be a temporal relation with $\mathcal{OIP}$ configuration (k, d, o). An *$\mathcal{OIP}$ partition*, $p_{i,j}$, with $0 \leq i \leq j < k$, spans all granules from i to j and has the partition interval $p_{i,j}.T = [o + i \cdot d, o + (j + 1) \cdot d - 1]$. A tuple $r \in \mathbf{r}$ is placed in partition $p_{i,j}$ iff $\lfloor \frac{r.T_S - o}{d} \rfloor = i$ and $\lfloor \frac{r.T_E - o}{d} \rfloor = j$.

Example 2. Relation $\mathbf{s}$ in Figure 2 includes seven tuples and spans the time range $U = [2012\text{-}1, 2012\text{-}12]$. The $\mathcal{OIP}$ configuration with $k = 4$ granules is $(4, 3, 2012\text{-}1)$ with granule duration $d = \lceil \frac{|U|}{k} \rceil = \lceil \frac{12}{4} \rceil = 3$ months and start time point $o = U_S =$ 2012-1. The partitions that span one granule are $p_{0,0}$, $p_{1,1}$, $p_{2,2}$, and $p_{3,3}$, each ranging over three months. The partitions $p_{0,1}$, $p_{0,2}$, $p_{0,3}$, $p_{1,2}$, $p_{1,3}$, and $p_{2,3}$ span more than one granule each, e.g., partition $p_{0,1}$ spans the range [2012-1, 2012-6]. Tuple s_1 is placed in partition $p_{0,0}$ since $\lfloor \frac{s_1.T_S - o}{d} \rfloor = \lfloor \frac{2012\text{-}1 - 2012\text{-}1}{3} \rfloor = \lfloor \frac{0}{3} \rfloor = 0$ and $\lfloor \frac{s_1.T_E - o}{d} \rfloor = \lfloor \frac{2012\text{-}1 - 2012\text{-}1}{3} \rfloor = \lfloor \frac{0}{3} \rfloor = 0$. Tuple s_6 is placed in partition $p_{1,3}$ since $\lfloor \frac{s_6.T_S - o}{d} \rfloor = \lfloor \frac{2012\text{-}6 - 2012\text{-}1}{3} \rfloor = \lfloor \frac{5}{3} \rfloor = 1$ and $\lfloor \frac{s_6.T_E - o}{d} \rfloor = \lfloor \frac{2012\text{-}10 - 2012\text{-}1}{3} \rfloor = \lfloor \frac{9}{3} \rfloor = 3$. Five out of ten partitions are empty, namely $p_{0,3}$, $p_{0,2}$, $p_{1,2}$, $p_{2,2}$, and $p_{3,3}$.

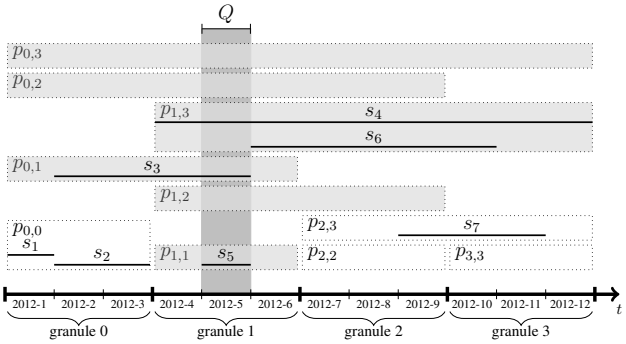


Figure 2: $\mathcal{OIP}$ with Configuration $(4, 3, 2012\text{-}1)$ for $\mathbf{s}$.

LEMMA 1 (*$\mathcal{OIP}$* OVERLAP QUERY) *Let (k, d, o) be an $\mathcal{OIP}$ configuration and $Q = [Q_S, Q_E]$ be a query interval. The candidate tuples that possibly overlap Q are in partitions $p_{i,j}$ for which $i \leq e = \lfloor \frac{Q_E - o}{d} \rfloor$ and $j \geq s = \lfloor \frac{Q_S - o}{d} \rfloor$. We term these partitions the* relevant partitions*; s is the start and e the end index of Q.*

PROOF. (By contradiction) Assume a partition $p_{i,j}$ with $j < s = \lfloor \frac{Q_S - o}{d} \rfloor$, which contains a tuple r that overlaps Q. According to Definition 2, tuple r is placed in a partition $p_{i,j}$ with $i = \lfloor \frac{r.T_S - o}{d} \rfloor$ and $j = \lfloor \frac{r.T_E - o}{d} \rfloor$. Since $j < s$, we get $r.T_E < Q_S$, i.e., $r.T$ and Q do not overlap, which contradicts our assumption. Similarly, a tuple in $p_{i,j}$ with $i > e = \lfloor \frac{Q_E - o}{d} \rfloor$ cannot overlap Q since $r.T_S > Q_E$. □

Example 3. Consider Figure 2 with the $\mathcal{OIP}$ configuration $(4, 3, 2012\text{-}1)$ and the query interval $Q = [2012\text{-}5, 2012\text{-}5]$. For the relevant partitions, $p_{i,j}$, the following constraints hold: $i \leq e = \lfloor \frac{Q_E - o}{d} \rfloor = \lfloor \frac{2012\text{-}5 - 2012\text{-}1}{3} \rfloor = \lfloor \frac{4}{3} \rfloor = 1$ and $j \geq s = \lfloor \frac{Q_S - o}{d} \rfloor = \lfloor \frac{2012\text{-}5 - 2012\text{-}1}{3} \rfloor = \lfloor \frac{4}{3} \rfloor = 1$. This is satisfied for partitions $p_{0,3}$, $p_{0,2}$, $p_{0,1}$, $p_{1,3}$, $p_{1,2}$, and $p_{1,1}$ (gray boxes), which contain all candidate tuples for the query interval Q.

Next, we establish the *constant clustering guarantee* of $\mathcal{OIP}$: the difference in duration between a tuple and its partition is less than two granules, i.e., constant and independent of the duration of a tuple. Note that the number of time points per granule or the duration of a granule have no impact on our solution. The constant clustering guarantee ensures (a) an excellent partitioning since the difference in duration between a tuple and its partitions is less than two granules, (b) an average false hit ratio that is independent of the intervals of tuples (cf. Section 5.1), and (c) it allows to take advantage of empty partitions by increasing k (cf. Section 6.2).

LEMMA 2 (CONSTANT CLUSTERING GUARANTEE) *Let (k, d, o) be an $\mathcal{OIP}$ configuration for relation $\mathbf{r}$. The difference in duration between a tuple $r \in \mathbf{r}$ and its partition p is less than $2d$:*

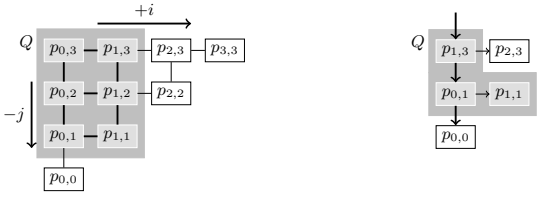
$$\forall r \in \mathbf{r}(r \in p \Rightarrow |p.T| - |r.T| < 2d).$$

PROOF. We show that the difference in duration of the smallest tuple in a partition $p_{i,j}$ and $p_{i,j}$ is less than $2d$. A tuple r is placed in partition $p_{i,j}$ iff $i = \lfloor \frac{r.T_S - o}{d} \rfloor$ and $j = \lfloor \frac{r.T_E - o}{d} \rfloor$ (cf. Definition 2). Thus, we have $i \cdot d \leq r.T_S - o \leq (i + 1)d - 1$ and $j \cdot d \leq r.T_E - o \leq (j + 1)d - 1$. The smallest tuple in $p_{i,j}$ has duration $|[(i + 1)d - 1, j \cdot d]| = j \cdot d - (i + 1)d + 2$, and $p_{i,j}$ has duration $|[i \cdot d, (j + 1)d - 1]| = (j + 1)d - i \cdot d$. Hence, the difference in duration between the smallest tuple r in $p_{i,j}$ and $p_{i,j}$ is $2d - 2 < 2d$. □

For instance, for a relation that spans 2000 days and with $k = 200$, we have $d = 10$ days. A tuple that spans 80 days can be in a partition that spans 90 days, which is a difference of 10 days. A tuple that spans 282 days can be in a partition that spans 300 days, which is a difference of 18 days. Note that the difference is always less than $2d = 20$ days.

4.2 Lazy Partitioning

We represent the $\mathcal{OIP}$ access structure as a triangle in a distance regular square grid graph [6], as illustrated in Figure 3(a) for the $\mathcal{OIP}$ in Figure 2. We call this a *triangular grid graph* with grid points (i, j) for $0 \leq i \leq j < k$. To find all relevant partitions for a query interval with start index s and end index e, we determine all partitions $p_{i,j}$ for which $j \geq s$ and $i \leq e$ (cf. Lemma 1). We start at the top-left corner of the grid (i.e., $i = 0, j = k - 1 = 3$) and move along the path with decreasing j as long as $j \geq s$. At each node $p_{0,j}$, we follow the path with increasing i as long as $i \leq min(j, e)$. In Figure 3(a), the relevant partitions (gray) for query interval Q are on the paths $p_{0,3} \rightarrow p_{1,3}$, $p_{0,2} \rightarrow p_{1,2}$ and $p_{0,1} \rightarrow p_{1,1}$.



(a) Triangular Grid Graph. (b) Lazy Partition List.

Figure 3: Management of $\mathcal{OIP}$ Partitions.

The number of possible $\mathcal{OIP}$ partitions corresponds to the number of nodes in a triangular grid graph and grows quadratically with the number of granules k.

PROPOSITION 1 (NUMBER OF PARTITIONS) *For k granules, the number of possible partitions is* $|\mathbf{p}| = \sum_{i=0}^{k-1}(k-i) = \frac{k^2+k}{2}$.

The *lazy partition list* is a compressed triangular grid graph that includes only non-empty partitions. Figure 3(b) shows the lazy partition list of our example. It includes only the non-empty partitions and the directed edges that are needed for navigation. The *main list* starts at the upper-left corner and connects nodes with decreasing j from top to bottom. Each node of the main list starts a *branch list* that connects (from left to right) nodes with the same j-value and increasing i-value.

The lazy partition list has the following advantages: (a) the number of CPU operations is reduced since empty partitions do not appear in the access structure; (b) k can be increased if not all partitions are used (cf. Section 5.2 and Section 6.2); and (c) the number of partitions is upper bounded by the cardinality of the partitioned relation, independent of the value of k.

LEMMA 3 (NUMBER OF PARTITIONS WITH LAZY PARTITIONING) *Assume an $\mathcal{OIP}$ configuration (k, d, o) for a relation $\mathbf{r}$ with n tuples whose valid time duration is at most λ. The number of partitions, $|\mathbf{p}|$, of $\mathcal{OIP}$ for $\mathbf{r}$ is upper bounded by* $min(k\lceil\lambda\cdot k\rceil + k - \frac{\lceil\lambda\cdot k\rceil^2}{2} - \frac{\lceil\lambda\cdot k\rceil}{2}, n)$.

PROOF. Tuples in $\mathbf{r}$ span at most $\lceil\lambda\cdot k\rceil$ granules. From Lemma 2 we have that the difference in duration of a partition and its tuples is less than two granules. Thus, the longest used partition spans at most $\lceil\lambda\cdot k\rceil + 1$ granules, and we have $|\mathbf{p}| \leq \sum_{x=0}^{\lceil\lambda\cdot k\rceil+1-1}(k-x) = k\lceil\lambda\cdot k\rceil + k - \frac{\lceil\lambda\cdot k\rceil^2}{2} - \frac{\lceil\lambda\cdot k\rceil}{2}$. Since empty partitions are not created, $|\mathbf{p}|$ cannot exceed n. □

Example 4. Assume a relation with tuple durations up to 20% of the relation's time range, i.e., $\lambda = 0.2$. With $k = 200$, at most $200\lceil 0.2\cdot 200\rceil + 200 - \frac{\lceil 0.2\cdot 200\rceil^2}{2} - \frac{\lceil 0.2\cdot 200\rceil}{2} = 7,380$ partitions out of $\frac{200^2+200}{2} = 20,100$ possible partitions are used, i.e., 37%, while 63% are empty.

4.3 Implementation of $\mathcal{OIP}$

Our implementation of $\mathcal{OIP}$ uses a lazy partition list, $\mathcal{L}$, to keep track of indices and storage blocks of partitions. Figure 4 shows the lazy partition list for our running example.

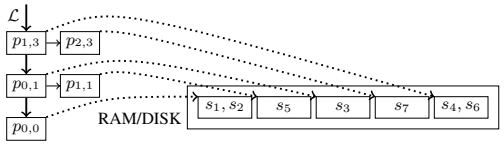


Figure 4: $\mathcal{OIP}$ Lazy Partition List $\mathcal{L}$ with Block Pointers.

Algorithm 1 creates the lazy partition list $\mathcal{L}$ for an input relation $\mathbf{r}$ with n tuples and an $\mathcal{OIP}$ configuration (k, d, o). After initializing an empty partition list, the relation is sorted according to the tuples' partition $p_{i,j}$ with j in ascending and i in descending order. The indices i and j of the partition to which a tuple r is assigned are computed according to Definition 2. The sorting ensures that tuples fall either into the first node of the list ($c = nil \vee c.j < j$) or into a new node that is prepended to $\mathcal{L}.head$ ($c.i > i$). The sorting reduces the complexity of insertions from $O(k)$ to $O(1)$ and gives a total runtime complexity for constructing $\mathcal{L}$ of $O(n \log n)$, which is independent of k and ensures that storage blocks are allocated sequentially.

Algorithm 1: OIPCREATE ($\mathbf{r}$, (k,d,o))

Input: Relation $\mathbf{r}$ and $\mathcal{OIP}$ configuration (k, d, o)
Output: Partition list $\mathcal{L}$

$\mathcal{L}$:= empty partition list;
Sort $\mathbf{r}$ by $\lfloor\frac{r.T_E-o}{d}\rfloor$ ASC and $\lfloor\frac{r.T_S-o}{d}\rfloor$ DESC;
foreach $r \in \mathbf{r}$ **do**
  $i := \lfloor\frac{r.T_S-o}{d}\rfloor$;
  $j := \lfloor\frac{r.T_E-o}{d}\rfloor$;
  $c := \mathcal{L}.head$;
  if $c = nil \vee c.j < j$ **then**
    $\mathcal{L}.head :=$ new node with partition $p_{i,j}$;
    $\mathcal{L}.head.down := c$;
  else if $c.i > i$ **then**
    $\mathcal{L}.head :=$ new node with partition $p_{i,j}$;
    $\mathcal{L}.head.down := c.down$;
    $\mathcal{L}.head.right := c$;
  Add r to $\mathcal{L}.head$;
return $\mathcal{L}$;

Example 5. Consider relation $\mathbf{s}$ in Figure 2. The call of OIPCREATE ($\mathbf{s}$, (4,3,2012-1)) constructs $\mathcal{L}$[2] as follows:

1. $\mathbf{r} = sort(\mathbf{s}) = \langle s_1, s_2, s_5, s_3, s_7, s_4, s_6\rangle$, $\mathcal{L} = \langle\rangle$
2. $r = s_1, i = \lfloor\frac{\text{2012-1}-\text{2012-1}}{3}\rfloor = 0, j = \lfloor\frac{\text{2012-1}-\text{2012-1}}{3}\rfloor = 0,$ $\mathcal{L} = \langle\langle(0,0,\{s_1\})\rangle\rangle$
3. $r = s_2, i = \lfloor\frac{\text{2012-2}-\text{2012-1}}{3}\rfloor = 0, j = \lfloor\frac{\text{2012-3}-\text{2012-1}}{3}\rfloor = 0,$ $\mathcal{L} = \langle\langle(0,0,\{s_1,s_2\})\rangle\rangle$
4. $r = s_5, i = \lfloor\frac{\text{2012-5}-\text{2012-1}}{3}\rfloor = 1, j = \lfloor\frac{\text{2012-5}-\text{2012-1}}{3}\rfloor = 1,$ $\mathcal{L} = \langle\langle(1,1,\{s_5\})\rangle, \langle(0,0,\{s_1,s_2\})\rangle\rangle$
5. $r = s_3, i = \lfloor\frac{\text{2012-2}-\text{2012-1}}{3}\rfloor = 0, j = \lfloor\frac{\text{2012-5}-\text{2012-1}}{3}\rfloor = 1,$ $\mathcal{L} = \langle\langle(0,1,\{s_3\}),(1,1,\{s_5\})\rangle, \langle(0,0,\{s_1,s_2\})\rangle\rangle$

The algorithm terminates and returns the lazy partition list $\mathcal{L} = \langle\langle(1,3,\{s_4,s_6\}),(2,3,\{s_7\})\rangle, \langle(0,1,\{s_3\}),(1,1,\{s_5\})\rangle, \langle(0,0,\{s_1,s_2\})\rangle\rangle$, which is illustrated in Figure 4.

5. ANALYTICAL RESULTS OF $\mathcal{OIP}$

In this section, we analyze the quality of $\mathcal{OIP}$ using two different measures. The *average false hit ratio*, AFR, measures the precision of a partitioning in terms of the average number of tuples that are retrieved for a query interval but do not contribute to the result. The *average number of partition accesses*, APA, quantifies the number of partitions that are fetched for a query.

5.1 Average False Hit Ratio

Definition 3. (*False Hits*) Let P be a partitioning of a valid-time relation $\mathbf{r}$ and Q be a query interval. The *false hits*, $\mathsf{F}(P, Q)$, are the tuples that are retrieved when fetching the relevant partitions, but are not part of the query result, i.e.,

$$\mathsf{F}(P,Q) = \{r \mid \exists p \in P\big(r \in p \wedge p.T \cap Q \wedge \neg\,(r.T \cap Q)\big)\}.$$

Consider Figure 2. For the query interval $Q = [\text{2012-5}, \text{2012-5}]$, only the relevant partitions are fetched (i.e., $p_{1,1}$, $p_{0,1}$, $p_{1,3}$). The false hits are $\mathsf{F}(\mathcal{OIP}, Q) = \{s_6\}$ since partition $p_{1,3}$ is fetched, but s_6 does not overlap Q. The result tuples are s_3, s_4, and s_5.

[2]We use nested lists as an abstract notation. For instance, $\mathcal{L} = \langle\langle a\rangle, \langle b, c\rangle\rangle$ has nodes a, b, c; $\mathcal{L}.head = a$; $a.down = b$; $b.right = c$; $a.right$, $b.down$, $c.down$, and $c.right$ are nil.

We proceed by defining the sum false hit ratio as the percentage of false hits over all possible point queries, i.e., the false hits produced by all queries over query intervals of duration 1 divided by the total number of tuples.

Definition 4. (*Sum False Hit Ratio*) Let P be a partitioning of a valid-time relation $\mathbf{r}$ with time range U. The *sum false hit ratio*, $\mathsf{SFR}(P)$, for all query intervals $[x, x]$ that overlap U is defined as

$$\mathsf{SFR}(P) = \frac{\sum_{x \in U} |\mathsf{F}(P, [x, x])|}{|\mathbf{r}|}.$$

For the $\mathcal{OIP}$ shown in Figure 2, we have $\mathsf{SFR}(\mathcal{OIP}) = \frac{|\mathsf{F}(\mathcal{OIP}, [\text{2012-1,2012-1}])| + \cdots + |\mathsf{F}(\mathcal{OIP}, [\text{2012-12,2012-12}])|}{7} = \frac{14}{7} = 2$. Thus, for all query intervals of duration 1, two times the number of tuples in $\mathbf{s}$ are retrieved as false hits.

LEMMA 4. *The sum false hit ratio of a partitioning P over a time range U is independent of the query interval duration q, i.e., it is the same for all query intervals $[x, x + q - 1]$ that overlap U for any value of q:*

$$\mathsf{SFR}(P) = \frac{\sum_{x \in U} |\mathsf{F}(P, [x, x])|}{|\mathbf{r}|} = \frac{\sum_{Q: Q \cap U \wedge |Q| = q} |\mathsf{F}(P, Q)|}{|\mathbf{r}|}.$$

PROOF. Consider a time point $x \in U$ and a partition $p \in P$. Query interval $[x, x]$ of duration 1 can produce false hits in p if there is a non-overlapping part before x (i.e., $p.T_S < x$) and/or a non-overlapping part after x (i.e., $x < p.T_E$). All tuples that start and end in one of the two non-overlapping parts are false hits. We consider now query intervals of duration $q > 1$. The query interval $[x, x+q-1]$ produces the same non-overlapping part before x, and the query interval $[x-q+1, x]$ the same non-overlapping part after x, yielding together exactly the same false hits for partition p as the point query with interval $[x, x]$. Since for each x there exists exactly one query interval of duration q that starts at x and one that ends at x, it is straightforward to generalize this result to the sum over all partitions and time points in U. This proves the lemma. □

Next, we define the average false hit ratio for an arbitrary query interval of duration $q \geq 1$.

Definition 5. (*Average False Hit Ratio*) Let P be a partitioning for a relation $\mathbf{r}$ with time range U. The *average false hit ratio*, $\mathsf{AFR}(P)$, for a query interval duration q is defined as

$$\mathsf{AFR}(P) = \frac{\mathsf{SFR}(P)}{|U| + q - 1}.$$

PROPOSITION 2. *The $\mathsf{AFR}(P)$ decreases monotonically with increasing query interval duration q.*

Example 6. Consider Figure 2 with the time range $U = [\text{2012-1, 2012-12}]$ and the sum false hit ratio $\mathsf{SFR}(\mathcal{OIP}) = 2$. The number of query intervals of duration $q = 1$ is $|U| + q - 1 = 12$, yielding an average false hit ratio $\mathsf{AFR}(\mathcal{OIP}) = 2 \cdot \frac{1}{12}$ $(= 16.7\%)$, i.e., on average 16.7% of the fetched tuples are false hits. The number of query intervals of duration $q = 5$ is $|U|+q-1 = 16$, yielding an average false hit ratio $\mathsf{AFR}(\mathcal{OIP}) = 2 \cdot \frac{1}{16}$ $(= 12.5\%)$.

For the analysis of the average false hit ratio of $\mathcal{OIP}$ in the following Theorem 1, we use duration complete relations. A *duration complete relation*, $\mathbf{r}^l_U$, contains exactly one tuple for each interval up to a duration l in the time range U, i.e.,

$$\forall T \subseteq U(|T| \leq l \Rightarrow \exists r \in \mathbf{r}^l_U(r.T = T)),$$
$$\forall r \in \mathbf{r}^l_U(|r.T| \leq l),$$
$$\forall r, r' \in \mathbf{r}^l_U(r \neq r' \Rightarrow r.T \neq r'.T).$$

For instance, the duration complete relation $\mathbf{r}^2_{[0,3]}$ contains a total of seven tuples with intervals $[0, 0]$, $[1, 1]$, $[2, 2]$, $[3, 3]$, $[0, 1]$, $[1, 2]$, $[2, 3]$. Duration complete relations ensure that the AFR is calculated over tuples of all possible positions and durations in U.

THEOREM 1. *Assume an $\mathcal{OIP}$ with configuration (k, d, o). The average false hit ratio for duration complete relations is independent of the duration of the tuples and upper bounded by*

$$\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}.$$

The proof of Theorem 1 is provided in the Appendix.

5.2 Average Number of Partition Accesses

The *average number of partition accesses*, APA, quantifies how many partitions are accessed on average to retrieve all tuples that overlap a query interval, i.e., how many relevant partitions exist.

LEMMA 5 (APA UPPER BOUND) *Assume an $\mathcal{OIP}$ with configuration (k, d, o), where all partitions are used. The average number of partition accesses for query intervals with uniformly distributed start and end time points is:*

$$\mathsf{APA}(\mathcal{OIP}) \leq \frac{k^2 + k + 1}{3}.$$

PROOF. For query intervals with uniformly distributed start and end time points, every query interval starting in granule s and ending in granule e has the same probability. Thus, we need to compute the number of partitions that a query interval accesses when starting in s and ending in e, which is the total number of partitions minus all partitions ending before s and all partitions starting after e as follows:

$$\begin{aligned} \#acc(s, e) &= \frac{k^2 + k}{2} - \sum_{i=0}^{s-1}(s - i) - \sum_{i=0}^{k-e-1}(k - e - 1 - i) \\ &= k + k \cdot e - \frac{s^2 + s}{2} - \frac{e^2 + e}{2}. \end{aligned}$$

We sum the number of partition accesses, $\#acc(s, e)$, for all $s \leq e < k$ and divide the sum by the cardinality of $s \leq e < k$ to get:

$$\mathsf{APA}(\mathcal{OIP}) = \frac{\sum_{e=0}^{k-1}\sum_{s=0}^{e}(\#acc(s, e))}{\sum_{e=0}^{k-1}\sum_{s=0}^{e}(1)} = \frac{k^2 + k + 1}{3}.$$

□

Since empty partitions are not present in the $\mathcal{OIP}$ access structure, APA is reduced if not all partitions are used. We use a tightening factor to quantify the reduction of partitions with lazy partitioning. The *tightening factor*, τ, with $0 < \tau \leq 1$, is calculated as the ratio between the number of used partitions with lazy partitioning (Lemma 3) and the total number of partitions (Proposition 1). For instance, the tightening factor in Example 4 is $\tau = 1890/5050 = 0.37$.

THEOREM 2 (APA) *Assume an $\mathcal{OIP}$ configuration (k, d, o) for a relation with n tuples and a tightening factor τ with $0 < \tau \leq 1$. The average number of partition accesses is:*

$$\mathsf{APA}(\mathcal{OIP}) \leq min(\tau \cdot \frac{k^2 + k + 1}{3}, n).$$

PROOF. The proof follows from Lemma 5 and Lemma 3. The tightening factor τ is the ratio of the number of used and the number of possible partitions. The inequality holds since the multiplication with τ conservatively assumes that the longest partitions, which produce more partition accesses than shorter partitions, are omitted. □

6. THE OVERLAP JOIN OIPJOIN

This section presents the OIPJOIN algorithm, derives the optimal number of granules k, illustrates its calculation by an example, and analyzes the runtime complexity of the OIPJOIN.

6.1 The OIPJOIN Algorithm

Algorithm 2 computes the OIPJOIN for relations $\mathbf{r}$ and $\mathbf{s}$. First, k (cf. Section 6.2) and the $\mathcal{OIP}$ configurations of the two relations are determined. The partitions are created by calling OIPCREATE, and the result relation $\mathbf{z}$ is initialized. Then, the algorithm iterates over each outer partition in $\mathcal{L}_r$ and performs an overlap query (cf. Lemma 1) with the query interval $[Q_S, Q_E]$ of the outer partition (cf. Definition 2). If $[Q_S, Q_E]$ does not overlap the time range of the inner relation $\mathbf{s}$, the outer partition is skipped. Otherwise, the indices s and e of the query interval $[Q_S, Q_E]$ are determined. The relevant partitions of the inner relation that overlap with the query interval are fetched, and the tuples are joined with the tuples in the outer partition. The result tuples are collected in $\mathbf{z}$.

Algorithm 2: OIPJOIN ($\mathbf{r}$, $\mathbf{s}$)

Input: Relation $\mathbf{r}$ and relation $\mathbf{s}$
Output: $\mathbf{z} = \{r \circ s \mid r \in \mathbf{r} \wedge s \in \mathbf{s} \wedge r.T \cap s.T\}$

Determine k for $\mathbf{r}$ and $\mathbf{s}$ for given IO and CPU costs;
Determine configurations (k, d_r, o_r) for $\mathbf{r}$ and (k, d_s, o_s) for $\mathbf{s}$;
$\mathcal{L}_r \leftarrow$ OIPCREATE($\mathbf{r}, (k, d_r, o_r)$);
$\mathcal{L}_s \leftarrow$ OIPCREATE($\mathbf{s}, (k, d_s, o_s)$);
$\mathbf{z} := \emptyset$;
foreach *node* c_r *in* $\mathcal{L}_r$ **do**
  $Q_S := o_r + c_r.i \cdot d_r$;
  $Q_E := o_r + (c_r.j + 1) \cdot d_r - 1$;
  if $Q_E \geq o_s \wedge Q_S < o_s + k \cdot d_s$ **then**
    $s := \lfloor \frac{Q_S - o_s}{d_s} \rfloor$;
    $e := \lfloor \frac{Q_E - o_s}{d_s} \rfloor$;
    $c_s := \mathcal{L}_s.head$;
    while $c_s \neq nil \wedge c_s.j \geq s$ **do**
      $x := c_s$;
      while $x \neq nil \wedge x.i \leq e$ **do**
        $\mathbf{z} := \mathbf{z} \cup \{$ joined tuples from c_r and $x\}$;
        $x := x.right$;
      $c_s := c_s.down$;
return $\mathbf{z}$;

Example 7. Consider Figure 1 with $k = 4$. We get the $\mathcal{OIP}$ configurations $(4, 2, \text{2012-5})$ for $\mathbf{r}$ and $(4, 3, \text{2012-1})$ for $\mathbf{s}$. OIPCREATE creates the lazy partition lists $\mathcal{L}_r = \langle\langle(1, 3, \{r_3\})\rangle, \langle(0, 0, \{r_1, r_2\})\rangle\rangle$ and $\mathcal{L}_s = \langle\langle(1, 3, \{s_4, s_6\}), (2, 3, \{s_7\})\rangle, \langle(0, 1, \{s_3\}), (1, 1, \{s_5\})\rangle, \langle(0, 0, \{s_1, s_2\})\rangle\rangle$. The first outer partition is processed as follows:

$c_r = (1, 3, \{r_3\})$
$Q_S = \text{2012-5} + 1 \cdot 2 = \text{2012-7}$
$Q_E = \text{2012-5} + (3 + 1) \cdot 2 - 1 = \text{2012-12}$
$s = \lfloor \frac{\text{2012-7} - \text{2012-1}}{3} \rfloor = \lfloor \frac{6}{3} \rfloor = 2$
$e = \lfloor \frac{\text{2012-12} - \text{2012-1}}{3} \rfloor = \lfloor \frac{11}{3} \rfloor = 3$
$c_s = \mathcal{L}_s.head = (1, 3, \{s_4, s_6\})$
$x = c_s = (1, 3, \{s_4, s_6\})$
$\mathbf{z} = \{r_3 \circ s_4, r_3 \circ s_6\}$
$x = x.right = (2, 3, \{s_7\})$
$\mathbf{z} = \{r_3 \circ s_4, r_3 \circ s_6, r_3 \circ s_7\}$
$c_s = c_s.down = (0, 1, \{s_3\})$

The second (and last) outer partition, $c_r = (0, 0, \{r_1, r_2\})$, is processed in a similar way, yielding the final result $\mathbf{z} = \{r_3 \circ s_4, r_3 \circ s_6, r_3 \circ s_7, r_1 \circ s_3, r_1 \circ s_4, r_1 \circ s_5, r_2 \circ s_4, r_2 \circ s_6\}$.

6.2 Number of Granules k

Choosing k (i.e., the number of granules) is the most important decision for the OIPJOIN. In order to derive k for the outer and the inner relation, we proceed in two steps. First, we provide a cost function for the OIPJOIN, and second, we minimize the cost function with respect to k.

Cost Function. The cost function considers the CPU cost of a comparison operation (c_cpu) and the cost of a block IO (c_io). A block IO can refer to either main memory or disk. The cost function models the *overhead* due to partition accesses and false hits. Recall that the cost for creating the partitioning is, due to sorting, independent of k and thus not included in the cost function.

Let k_r and k_s be the number of granules for the outer and inner relation, respectively. For the join we fetch, for each of the $O(k_r^2)$ outer partitions, $O(k_s^2)$ inner partitions, i.e., $O(k_r^2 \cdot k_s^2)$. Furthermore, for each outer and inner tuple we have, respectively, $O(\frac{n_s}{k_s})$ and $O(\frac{n_r}{k_r})$ false hits, i.e., $O(n_s \cdot \frac{n_r}{k_r} + n_r \cdot \frac{n_s}{k_s})$. Both, $O(k_r^2 \cdot k_s^2)$ and $O(n_s \cdot \frac{n_r}{k_r} + n_r \cdot \frac{n_s}{k_s})$ reach their minimum when $k_r = k_s$, i.e., outer and inner relation are partitioned using the same number of granules k. Thus, we have a cost function with $k = k_r = k_s$:

$$\begin{aligned} cost(k) &= |\mathbf{p}_r| \cdot \mathsf{APA} \cdot (c\_io + 2 \cdot c\_cpu) + \\ &\quad |\mathbf{p}_r| \cdot n_s \cdot \mathsf{AFR} \cdot (\frac{c\_io}{b} + 2 \cdot \frac{n_r}{|\mathbf{p}_r|} \cdot 2 \cdot c\_cpu) \quad (1) \\ &= x \cdot \mathsf{APA} + y \cdot \mathsf{AFR} \end{aligned}$$

The first line is the cost for partition accesses. For each of the $|\mathbf{p}_r|$ outer partitions, the algorithm accesses APA inner partitions. Each partition access costs one extra c_io since an inner partition can have at most one partially filled block (remember we only measure the overhead) and two c_cpu (comparison i and j) for checking if this partition in the lazy partition list is relevant. The second line is the cost for false hits. For each of the $|\mathbf{p}_r|$ outer partitions, $n_s \cdot \mathsf{AFR}$ false hits in the relevant inner partitions produce extra block transfers, where b is the average number of tuples per block of the inner relation. The costs for identifying false hits is two c_cpu (comparison T_S and T_E) for each false hit in the outer partition and each false hit in the relevant inner partitions.

Determining k. We derive k by minimizing the cost function using the partial derivative of $x \cdot \mathsf{APA} + y \cdot \mathsf{AFR}$. The terms quantify, respectively, the increase of the costs due to partition accesses and the decrease of the costs due to false hits. k can be increased as long as the costs for AFR decrease more than the costs for APA increase. The optimal k is the point where the cost for accessing partitions starts growing faster than the cost for false hits decreases, which is the minimum of the cost function.

Since the complexity of the cost function prevents an analytical solution of the minimization problem, we proceed in two steps to derive k. First, we keep $|\mathbf{p}_r|$ and τ constant and derive k as follows:

1. Compute the partial derivative of $x \cdot \mathsf{APA} + y \cdot \mathsf{AFR}$. We use APA and AFR from Theorems 1 and 2 to get $x \cdot \tau \cdot \frac{k^2+k+1}{3} + y \cdot \frac{1}{k}$. The partial derivative with respect to k is $\partial_k(x \cdot \tau \cdot \frac{k^2+k+1}{3} + y \cdot \frac{1}{k}) = x \cdot \tau \cdot (\frac{2}{3} \cdot k + \frac{1}{3}) - \frac{y}{k^2}$.
2. Solve $x \cdot \tau \cdot (\frac{2}{3} \cdot k + \frac{1}{3}) - \frac{y}{k^2} = 0$ to get the k that minimizes the cost function:

$$k = \frac{\sqrt[3]{(162 \cdot y - x \cdot \tau + 18 \cdot \sqrt{y \cdot (81 \cdot y - x \cdot \tau)}) \cdot (x \cdot \tau)^2}}{6 \cdot x \cdot \tau} + \\ + \frac{x \cdot \tau}{3(\sqrt[3]{(162 \cdot y - x \cdot \tau + 18 \cdot \sqrt{y \cdot (81 \cdot y - x \cdot \tau)}) \cdot (x \cdot \tau)^2} \\ - \frac{1}{6} \approx \sqrt[3]{\frac{3 \cdot y}{2 \cdot x \cdot \tau}}.$$

In the second step, we use an iterative process that refines $|\mathbf{p}_r|_n$ and τ_n in each step in order to determine k. More specifically, we calculate the number $|\mathbf{p}_r|_n$ of outer partitions and the tightening factor τ_n from the previously calculated k_n, starting with $k_0 = 1$. After substituting x and y (cf. Equation (1)) in the above equation for k, we obtain the recurrence:

$$k_{n+1} = \sqrt[3]{\frac{3 \cdot n_s}{2 \cdot (c\_io + 2 \cdot c\_cpu) \cdot \tau_n} \cdot \left(\frac{c\_io}{b} + \frac{4 \cdot n_r \cdot c\_cpu}{|\mathbf{p}_r|_n}\right)} \quad (2)$$

We start with $k_0 = 1$ and calculate the number of outer partitions, $|\mathbf{p}_r|_0$, according to Lemma 3, i.e.,

$$|\mathbf{p}_r|_n = min(k_n \lceil \lambda_r \cdot k_n \rceil + k_n - \frac{\lceil \lambda_r \cdot k_n \rceil^2}{2} - \frac{\lceil \lambda_r \cdot k_n \rceil}{2}, n_r),$$

and the tightening factor τ_0 as the number of inner partitions (cf. Lemma 3) divided by the number of possible partitions (cf. Proposition 1), i.e.,

$$\tau_n = \frac{min(k_n \lceil \lambda_s \cdot k_n \rceil + k_n - \frac{\lceil \lambda_s \cdot k_n \rceil^2}{2} - \frac{\lceil \lambda_s \cdot k_n \rceil}{2}, n_s)}{(k_n^2 + k_n)/2}.$$

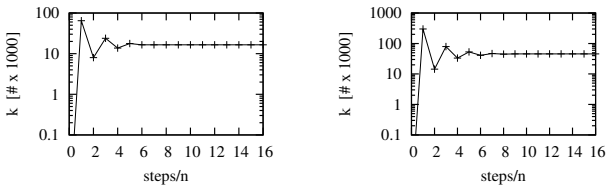
We repeatedly calculate k_{n+1} from $|\mathbf{p}_r|_n$ and τ_n until k converges to the minimum cost.

Example 8. Consider two relations $\mathbf{r}$ and $\mathbf{s}$, each with a time range $|U| = 10$M. Relation $\mathbf{r}$ has $n_r = 10$M tuples, and the maximum duration of tuples is $l_r = 1,000$, i.e., $\lambda_r = 0.0001$. Relation $\mathbf{s}$ has $n_s = 100$M tuples, and the maximum duration of tuples is $l_s = 5,000$, i.e., $\lambda_s = 0.0005$. Both relations are stored in main memory in blocks of 512 bytes. With a tuple size of 35 bytes, $b = 14$ tuples fit into a block. The time of a CPU operation is 0.5 nsec (2GHz), and the time to fetch a block from main memory is 10 nsec, i.e., $c\_cpu = 0.5$ and $c\_io = 10$. Starting with $n = 0$ and $k_0 = 1$, we get the following values:

n	k_n	$\lvert\mathbf{p}_r\rvert_n$	τ_n
0	1	1	1
1	64,633	517,036	0.00105
2	7,967	15,933	0.00126
3	23,819	95,270	0.00109
4	13,761	41,280	0.00116
5	17,795	53,382	0.00112
6	16,522	49,563	0.00121
7	**16,521**	**49,560**	**0.00121**
8	**16,521**	**49,560**	**0.00121**

Thus, k converges to $k = 16,521$, which is the number of granules for the OIPJOIN.

Figure 5 illustrates the convergence of k for relations of different size. The iterative process to find k converges since at each step we reduce the power by $\frac{1}{3}$. Note that due to the ceiling function and integer calculus in $|\mathbf{p}_r|_n$ and τ_n, k may not converge to a single number, but oscillate between two. In this case the final k is the average between these two numbers.



(a) $n_r = 10$M and $n_s = 100$M (b) $n_r = 100$M and $n_s = 1$G

Figure 5: Convergence of k.

6.3 Complexity Analysis

The complexity of the OIPJOIN is composed of three parts: $O(|\mathbf{p}_r| \cdot \mathsf{APA})$ partition fetches; $O(n_s \cdot n_r \cdot \mathsf{AFR})$ false hits; and $O(n_z)$ for retrieving n_z result tuples. After substituting AFR and APA according to Theorem 1 and 2, we get the sum $O(|\mathbf{p}_r| \cdot \tau \cdot k^2) + O(n_s \cdot n_r \cdot \frac{1}{k}) + O(n_z)$. The asymptotic k according to Equation (2) is $k = O((\frac{n_s \cdot n_r}{|\mathbf{p}_r| \cdot \tau})^{1/3})$.

The *upper bound complexity* occurs with tightening factor $\tau = 1$ (no tightening). In this case we get a low k to keep the cost for partition accesses low. From $\tau = 1$ we get $|\mathbf{p}_r| = O(k^2)$ (cf. Section 5.2), i.e.,

$$k = O((\frac{n_s \cdot n_r}{k^2})^{1/3})$$
$$k^{5/3} = O((n_s \cdot n_r)^{1/3})$$
$$k = O((n_s \cdot n_r)^{1/5})$$

Inserting this into the above sum gives $O(n_r^{4/5} \cdot n_s^{4/5} + n_z)$.

The *lower bound complexity* occurs with tightening factor $\tau = O(\frac{1}{k})$ (maximal tightening). In this case we get a high k, since the cost for partition accesses is low. From $\tau = O(\frac{1}{k})$ we get $|\mathbf{p}_r| = O(k)$. Then k is:

$$k = O((\frac{n_s \cdot n_r}{k \cdot \frac{1}{k}})^{1/3}) = O((n_s \cdot n_r)^{1/3})$$

Inserting this into the sum gives $O(n_r^{2/3} \cdot n_s^{2/3} + n_z)$.

To illustrate the complexity, we performed an overlap join between two relations with 5M tuples each and between two relations with 10M tuples each. As a reference point, we also compare it with the lower and upper bound of a sort-merge join (SMJ) of the same relations. Table 1 shows the runtimes in seconds (as usual, the time to write result tuples is excluded). We can see that the runtime increased approximately by a factor of $2^{2/3} \cdot 2^{2/3} = 2.52$ for the lower bound and by a factor of $2^{4/5} \cdot 2^{4/5} = 3.03$ for the upper bound. The increase in runtime for the sort-merge join is 2.06 (linear) for the lower bound and 4.00 (quadratic) for the upper bound.

		5M	10M	increase
OIPJOIN:	LB ($\tau \approx 1/k$)	46	120	×2.61
	UB ($\tau = 1$)	2,028	6,655	×3.28
SMJ:	LB	3.2	6.6	×2.06
	UB	81,043	324,175	×4.00

Table 1: Runtime and Factor of Runtime Increase.

7. EMPIRICAL EVALUATION

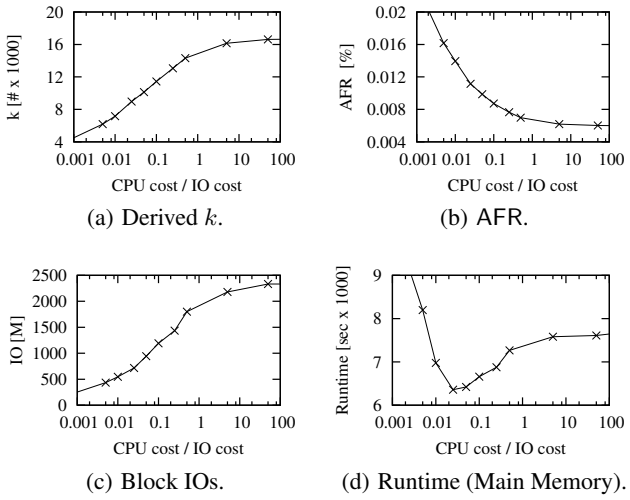
This section evaluates the performance of the OIPJOIN and compares it empirically with the other self-adjusting approaches. The first set of experiments evaluates how k adapts to the cost of CPU

operations and the cost of block IOs. We also verify our cost function by relating it to the actual runtime. The second set of experiments evaluates the performance of different approaches for a varying percentage and distribution of long-lived tuples. The ability to efficiently process data with long-lived tuples, i.e., tuples with a non-negligible temporal duration, is the most crucial aspect of algorithms and access methods for temporal data. The OIPJOIN outperforms related approaches by a large margin. The third set of experiments shows that the OIPJOIN scales better than the other approaches for real world datasets, coming from animal feed industry, personnel office, and open source software. Between $0.03\% - 20\%$ of the tuples are larger than 8% of the data's time range, which already leads to significant differences. Finally, we show that the OIPJOIN scales better than the other approaches for disk resident data since it considers both CPU and IO costs.

Setup. For the experiments we use a 2 x Intel(R) Xeon(R) CPU E5-2440 0 @ 2.40GHz machine with 64GB main memory running CentOS 6.4. All algorithms have been implemented in C. We use a tuple size of 35 bytes. The block size is 512 bytes for relations stored in main memory (gives the best performance on our machine) and 4K bytes (physical disk block size) when stored on disk. We implemented all algorithms for both disk and main memory storage. The cost to perform a CPU operation (0.5 nsec) on our machine is about 20 times faster than fetching a main memory block (10 nsec). We use synthetic datasets with a time range of $[1, 2^{24}]$ as well as real world datasets (described below).

The OIPJOIN is compared against the following state-of-the-art approaches. *Loose quadtree (lqt)*: We implemented a partition-based algorithm that joins every node of the outer tree with all relevant nodes in the inner tree. We use a cell expansion factor $p = 1$, which is widely accepted as the best value [18, 23] and gave the best results in our experiments. We use density based splitting, i.e., tuples are propagated down the tree only if a block is full. Together with block storage, this gave a runtime improvement up to 400% compared to random access to single tuples. Since in all experiments the loose quadtree outperformed the quadtree, the latter is not shown in the plots. *Relational interval tree (rit)*: We implemented the RI-Tree Up-Down partition-based algorithm [9]. When data is stored in main memory, we do not use blocks to store tuples contiguously. The reason is that even for a clustering index, the time to fetch 512 bytes that contain only a few matching tuples outweighs the advantages of contiguous memory access. *Segment tree (sgt)*: We implemented the segment tree, where the index is build on the inner relation and the overlap join is computed by joining each tuple of the outer relation with the segment tree. Duplicates are identified during join processing by testing whether the intersecting interval starts before the currently joined segment; if so, it has already been joined in a previous segment. *Sort-merge join (smj)*: We implemented a sort-merge join that sorts the tuples of the outer relation by the end point and the tuples of the inner relation by the start point. The sort order of the inner relation is used to stop scanning when an inner tuple has a larger start point than the end point of the outer tuple. The sort order of the outer relation allows to limit the backtracking to the maximum duration of tuples in the relations. We implemented the join using blocks. In spite of more false hits, this increases the performance due to less backtracking. All runtime experiments include the time to create the indices. For all approaches, the index creation time is $\approx 1\%$ of the total runtime for data kept in memory and $\approx 5\%$ for disk resident data.

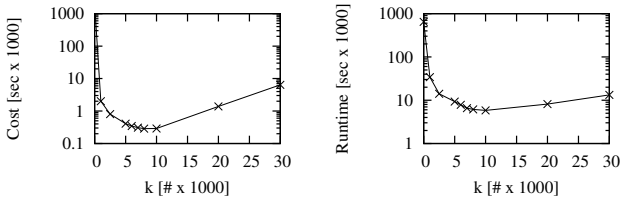
Number of Granules k. The first experiment shows how the OIPJOIN adapts to *c_cpu* and *c_io*. We use synthetic data: an outer relation with 10M tuples and an inner relation with 100M tuples, both with tuple durations up to 0.1% of the time range. Figure 6(a) shows k when varying the ratio $\frac{c\_cpu}{c\_io}$ from 0.001 to 100. When *c_cpu* gets more expensive, k increases so that more partitions are generated. Figure 6(b) and 6(c) show, respectively, the corresponding AFR (decreasing) and the number of block IOs (increasing). Figure 6(d) shows the runtime for main memory resident data. It illustrates that also for data that is stored in main memory the performance can be increased if the costs of memory IOs and the costs of CPU operations are considered for determining the optimal k.



(a) Derived k. (b) AFR. (c) Block IOs. (d) Runtime (Main Memory).

Figure 6: Derived k with Varying *c_cpu* and *c_io*.

The next experiment compares the cost function of the OIPJOIN to the actual runtime. We use the same relations as in the previous experiment and vary k. Figure 7(a) shows the cost function for *c_cpu* $= 0.5$ nsec and *c_io* $= 10$ nsec. Figure 7(b) shows the actual runtime for the same setting. It is easy to see that both functions have the same shape with the minimum at $k = 10, 130$.

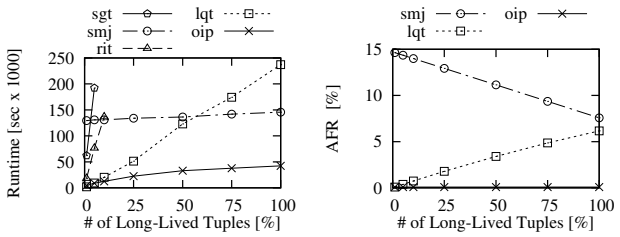


(a) Cost Function (Overhead). (b) Runtime.

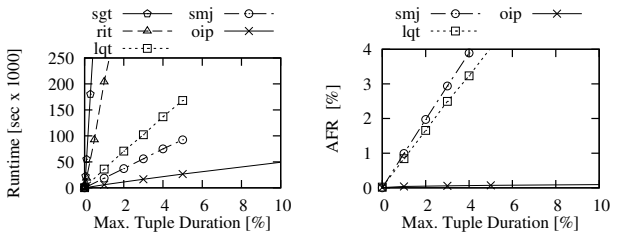
Figure 7: Cost Function and Runtime.

Long-Lived Tuples. The next experiment compares the performance of the OIPJOIN (oip) with the loose quadtree (lqt), the relational interval tree (rit), the segment tree (sgt), and the sort-merge join (smj), by varying the number of long-lived tuples and the maximum duration of tuples. The two input relations have 10M tuples each, with long-lived tuples that have a duration up to 8% and an average duration of 4% of the relation's time range. Short-lived tuples have a maximum duration of 0.01% of the time range. Figure 8 shows the runtime and the AFR of the four algorithms. The AFR of the relational interval tree and segment tree are omitted since they produce no false hits. The OIPJOIN significantly outperforms the

other approaches since it does not suffer from long-lived tuples and has a very small AFR (the curve is close to the x-axis). In contrast, the loose quadtree is very sensitive to long-lived tuples, and the AFR increases drastically. This yields much higher runtimes due to excessive comparison operations and the filtering of false hits. Although the relational interval tree does not produce false hits, its performance decreases with the increase of long-lived tuples since a higher number of index nodes need to be joined, which requires a high number of operations on the indices. The segment tree scales worse than the relational interval tree, since with longer tuple durations many duplicates need to be fetched and tested. In our experiments, the segment tree outperforms the relational interval tree only for tuple durations smaller than 0.001%. The performance of the sort-merge join is highly affected by the longest tuple in the dataset, but it scales better than the loose quadtree.



(a) Varying Number of Long-Lived Tuples.



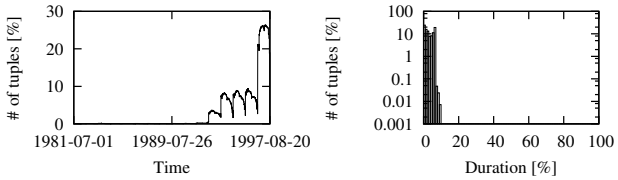
(b) Varying Maximum Duration of Tuples.

Figure 8: Long-Lived Tuples.

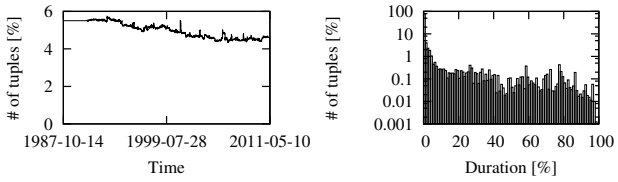
Real World Datasets. We use three real world datasets that differ in size and data distribution. The main properties of these datasets are summarized in Table 2. The Incumbent dataset [12] records the history of employees assigned to projects over a 16 year period at a granularity of days. The Feed dataset records the history of measured nutritive values of feeds over a 24 year period at a granularity of days; a measurement of a nutrient remains valid until a new measurement for the same nutritive value and feed becomes available. The Webkit dataset [1] records 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. Figure 9 shows the temporal distributions of the data (i.e., the number of overlapping tuple intervals at each time point) and the histograms of tuple durations.

	Incumbent	Feed	Webkit
Cardinality	$83,852$	$3,697,957$	$1,213,476$
Time Range	$5,895$	$8,610$	$\approx 2^{39}$
Min. Duration	1	1	$\approx 2^{10}$
Max. Duration	574	$8,589$	$\approx 2^{39}$
Avg. Duration	184	432	$\approx 2^{34}$
Distinct Points	$2,689$	$5,584$	$110,165$

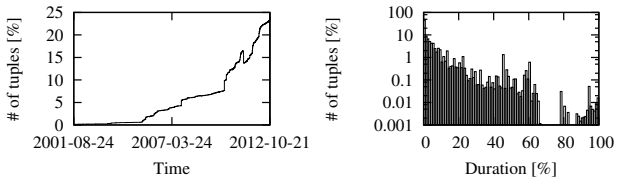
Table 2: Properties of Real World Datasets.



(a) Incumbent Dataset.



(b) Feed Dataset.

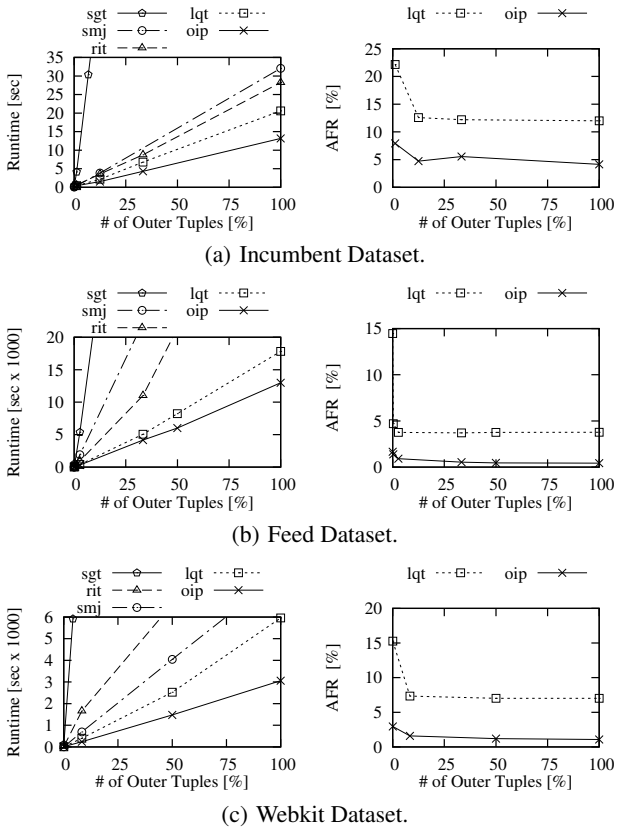


(c) Webkit Dataset.

Figure 9: Tuple Intervals per Time Point and Duration Histogram of Real World Datasets.

For all three datasets we perform an overlap join, using a subset of the dataset as the outer relation and the entire dataset as the inner relation. We use the smaller as the outer relation, since it typically has fewer partitions, and thus some partitions of the larger relation are not accessed at all during the join. Figure 10 shows the runtime and the AFR for the three datasets depending on the size of the outer relation. The OIPJOIN has the best performance in all three settings. The other approaches suffer from long-lived tuples, e.g., the AFR of the loose quadtree is much larger than the one of the OIPJOIN, and it does not adapt to the size of the dataset. The AFR of the sort-merge join is omitted since it reaches 30–50%.

Scalability on Disk. The last experiment shows the scalability of the algorithms for disk resident data. We vary the number of inner tuples from 100M to 1500M. The number of outer tuples is 1% of the inner relation. Both relations have tuple durations up to 0.1% of the time range. c_io is 200 times higher than c_cpu. Figures 11(a) and 11(b) show the number of block IOs and the AFR. Figure 11(c) shows the runtime behavior on a server with 64GB of main memory, where a large number of disk blocks is cached by the operating system. Although the loose quadtree, due to its density-based splitting strategy, is the best approach in terms of block IOs, it produces a large number of false hits. The OIPJOIN adapts to both the cost of block IOs and the cost of false hits, and thus outperforms all other approaches in terms of runtime. The segment tree performs worst, in particular in terms of IO (close to the y-axis), since for each outer tuple, duplicated inner tuples and thus disk blocks are fetched several times. We run the same experiment for the three best approaches on a different machine with a similar CPU but only 4GB main memory, that is, fewer disk blocks are cached by the operating system. The runtime behavior is shown in Figure 11(d) and is slower, as expected. The loose quadtree per-

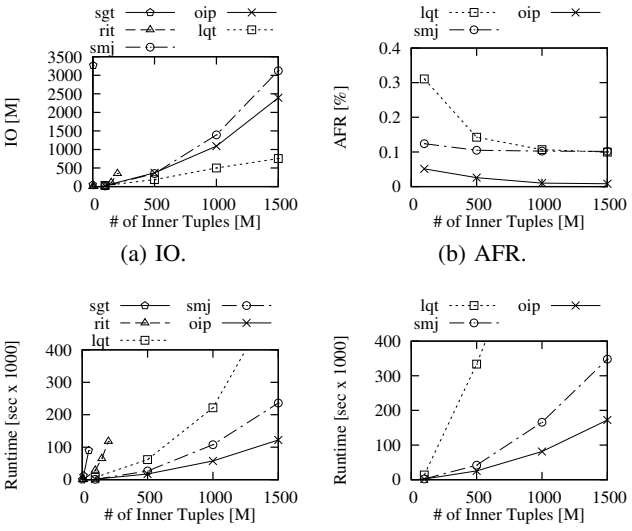


(a) Incumbent Dataset.

(b) Feed Dataset.

(c) Webkit Dataset.

Figure 10: Runtime and AFR for Real World Datasets.



(a) IO.

(b) AFR.

(c) Runtime on 64GB Server.

(d) Runtime on 4GB Server.

Figure 11: Varying Number of Tuples of Disk Resident Data.

forms much worse on this machine despite fewer IOs. The reason is that the OIPJOIN and the sort-merge join benefit from sequential reads due to sorting. The loose quadtree does not have sequential blocks on disk, hence the disk seek time deteriorates the performance of the loose quadtree.

Summary. The OIPJOIN is the most efficient and robust approach if the data includes long-lived tuples since it provides a constant clustering guarantee and adapts to both the cost for false hits and the cost for partition accesses. The loose quadtree and the relational interval tree are only competitive if the dataset contains a very low number of long-lived tuples. In all other cases, either the false hits or the navigation in the index structure incur high costs. For datasets with only very short tuples (or point data), the sort-merge join is the most efficient approach, but it deteriorates as soon as the dataset contains a few long-lived tuples.

8. CONCLUSION

In this paper, we have presented the overlap interval partition join (OIPJOIN) for valid-time relation together with *Overlap Interval Partitioning* ($\mathcal{OIP}$). $\mathcal{OIP}$ permits overlapping partitions that are not derived from a recursive hierarchical space division. In contrast to other approaches, the OIPJOIN does not deteriorate in the presence of long-lived tuples and adjusts the number of partitions based on the size of the dataset, the cost of CPU operations, and the cost of IOs. An in-depth empirical evaluation shows that the OIPJOIN outperforms state-of-the-art techniques for the overlap join.

Future work points in several directions. First, it is interesting to investigate how to update $\mathcal{OIP}$ incrementally if the relation changes, since the partitioning allows an expansion on both space boundaries by increasing k and maintaining an offset on the indices. For this the effects on k and the treatment of the ending variable "now" must be studied. Second, it is possible to refine the cost function for, e.g., different buffer replacement strategies. Third, we have planned to develop statistics to tighten k not only based on the maximum duration of tuples, but also on the data distribution.

9. REFERENCES

[1] The webkit open source project. `http://www.webkit.org`, 2012.

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

[3] N. Beckmann and B. Seeger. A revised R*-tree in comparison with related index structures. In *SIGMOD*, pages 799–812, 2009.

[4] J. L. Bentley. Solutions to klee's rectangle problems. Technical report, Carnegie Mellon University, 1977.

[5] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. More geometric data structures. In *Computational Geometry*, pages 211–233. Springer Berlin Heidelberg, 2000.

[6] A. E. Brouwer, A. M. Cohen, and A. Neumaier. *Distance Regular Graphs*. Springer-Verlag, 1989.

[7] A. Dignös, M. Böhlen, and J. Gamper. Query time scaling of attribute values in interval timestamped databases. In *ICDE*, pages 1304–1307, 2013.

[8] A. Dignös, M. H. Böhlen, and J. Gamper. Temporal alignment. In *SIGMOD*, pages 433–444, 2012.

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

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

[11] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.

[12] J. A. G. Gendrano, R. Shah, R. T. Snodgrass, and J. Yang. University information system (uis) dataset. TimeCenter CD-1, 1998.

[13] N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.

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

[15] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258, 1996.

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

[17] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. Touch: In-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.

[18] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[19] H. Samet, J. Sankaranarayanan, and M. Auerbach. Indexing methods for moving object databases: games and other applications. In *SIGMOD*, pages 169–180, 2013.

[20] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[21] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *ICDE*, pages 282–292, 1994.

[22] V. J. Tsotras and N. Kangerlaris. The snapshot index: An i/o-optimal access method for timeslice queries. *Inf. Syst.*, 20(3):237–260, 1995.

[23] T. Ulrich. Loose octrees. In *Game Programming Gems*, pages 444–453. Charles River Media, 2000.

APPENDIX

A. PROOF OF THEOREM 1

PROOF. (Theorem 1) The proof is split into three parts. In Part 1 we compute the SFR of $\mathcal{OIP}$ for a duration complete relation $\mathbf{r}^l_{kd}$ and a query interval with duration q. In Part 2 we use the result of Part 1 to show that $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$ for tuples of duration $l = 1$ and query intervals of duration $q = 1$. From Proposition 2 we have that the $\mathsf{AFR}(\mathcal{OIP})$ for $q > 1$ is smaller. In Part 3 we show that for tuples up to larger durations, i.e., $l > 1 \leq k \cdot d$, the SFR of $\mathcal{OIP}$ and the $\mathsf{AFR}(\mathcal{OIP})$ are smaller than for $l = 1$, thus $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$.

Part 1: Assume $l \leq d$. We compute first the sum of false hits for partitions p that span one granule. Assume a query interval Q with $q = 1$ that overlaps p. Let $v \geq 0$ be the duration of the non-overlapping part of p before Q starts. If $v \leq l$, we have $\frac{v^2+v}{2}$ false hits in this part of the partition, i.e., all intervals in p that start and end before Q. If $v \geq l$, we have $\frac{v^2+v}{2} - \sum_{x=l}^{v}(v-x) = vl - \frac{l^2-l}{2}$ false hits in this part, i.e., all intervals in p up to duration l that start and end before Q. We sum the false hits of all query intervals of duration 1 and get $\sum_{v=0}^{l-1}(\frac{v^2+v}{2}) + \sum_{v=l}^{d-1}(vl - \frac{l^2-l}{2}) = \frac{l^3-l}{6} + \frac{ld^2-l^2d}{2}$. The same sum is obtained for false hits in the non-overlapping part after the query interval Q ends. Thus, for k granules we get a total of $k(\frac{l^3-l}{3} + ld^2 - l^2d)$ false hits.

Next, we compute the sum of false hits for partitions that span more than one granule. For $l \leq d$, only partitions of duration $2d$ contain tuples. Each of these partitions contains $\frac{l^2-l}{2}$ tuples, i.e., all tuples up to duration l that start in the first half and end in the second half of the partition. The total number of matching tuples for all query intervals of duration 1 is $2\sum_{p=1}^{l-1}\sum_{x=1}^{p}(x) = \frac{l^3-l}{3}$. Thus, for $k-1$ partitions of duration $2d$ there are $2d$ possible query intervals of duration $q = 1$ that overlap a partition with $\frac{l^2-l}{2}$ tuples. Subtracting from these tuples the $\frac{l^3-l}{3}$ matches gives a total of $(k-1)(2d\frac{l^2-l}{2} - \frac{l^3-l}{3})$ false hits.

Adding up the false hits for the partitions and dividing the sum by the number of tuples $|\mathbf{r}^l_{kd}| = kdl - \frac{l^2-l}{2}$ we get

$$\mathsf{SFR}(\mathcal{OIP}) = \frac{2(l^2 - 3dl + 3kd^2 - 3kd + 3d - 1)}{3(2kd - l + 1)} \text{ for } l \leq d. \quad (3)$$

Now assume $l > d$. Let l be a multiple of d and $h = l/d$. Partitions that span one granule are completely full. This yields a total of $k\frac{d^3-d}{3}$ false hits, by replacing l in the first case of the proof with d. Then we have $\sum_{x=1}^{h-1}(k-x)$ partitions that span more than one granule, are not longer than l, and are completely filled. Each of these partitions produces up to $d^3 - d^2$ false hits. The only partitions that are longer than l and contain tuples of duration $l = hd$ have duration $d(h+1)$, of which $(k-h)$ exist. Each of these partitions contains $\frac{d^2-d}{2}$ tuples up to size l. The total number of matching tuples for all query intervals of duration 1 is $2\sum_{p=1}^{d-1}\sum_{x=1}^{p}(x) = \frac{d^3-d}{3}$, and the total number of matches where no false hits are possible is $\frac{d(h-1)(d^2-d)}{2}$. Thus, for $k-h$ partitions we get $(k-h)\Big(d(h+1)\frac{d^2-d}{2} - \big(\frac{d^3-d}{3} + \frac{d(h-1)(d^2-d)}{2}\big)\Big)$. Finally, we divide the sum by the number of tuples $|\mathbf{r}^l_{kd}| = kdl - \frac{l^2-l}{2}$ and get

$$\mathsf{SFR}(\mathcal{OIP}) = \frac{(d-1)(6kd - d + 2 - 3l)}{3(2kd - l + 1)} \text{ for } l > d. \quad (4)$$

Part 2: We show that $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$ for tuples of duration $l = 1$ and query intervals of duration $q = 1$. We use the $\mathsf{SFR}(\mathcal{OIP})$ of Part 1 for $l \leq d$ (since d must be at least 1) and set $l = 1$ in Equation (3) to get:

$$\mathsf{SFR}(\mathcal{OIP}) = \frac{2(1^2 - 3d1 + 3kd^2 - 3kd + 3d - 1)}{3(2kd - 1 + 1)} = d - 1$$

Next, we set $q = 1$ and divide by the number of query intervals $kd + q - 1 = kd + q - 1 = kd$ (ref. Definition 5), and get

$$\mathsf{AFR}(\mathcal{OIP}) = \frac{\mathsf{SFR}(\mathcal{OIP})}{kd + q - 1} = \frac{d-1}{kd + 1 - 1} = \frac{1}{k} - \frac{1}{kd} < \frac{1}{k}.$$

Part 3: We show that for $l > 1 \leq kd$ the $\mathsf{SFR}(\mathcal{OIP})$ is smaller than for $l = 1$, i.e., smaller $d - 1$ and thus $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$. Recall that the SFR is independent of the query interval duration q (ref. Lemma 4). First, we consider $1 < l \leq d$ and Equation (3):

$$\begin{aligned} d - 1 &> \frac{2(l^2 - 3dl + 3kd^2 - 3kd + 3d - 1)}{3(2kd - l + 1)} \\ 3(2kd - l + 1)(d - 1) &> 2(l^2 - 3dl + 3kd^2 - 3kd + 3d - 1) \\ -2l^2 + (3 + 3d)l - 3d - 1 &> 0 \end{aligned}$$

By solving the quadratic equation $-2l^2 + (3+3d)l - 3d - 1 = 0$ we get $l = \{1, \frac{3d+1}{2}\}$, since the quadratic term is negative we have an concave down parabola and thus the inequality we need to show for $1 < l \leq d$ holds for $1 < l < \frac{3d+1}{2}$.

Second, we consider $d < l \leq kd$ and Equation (4):

$$\begin{aligned} d - 1 &> \frac{(d-1)(6kd - d + 2 - 3l)}{3(2kd - l + 1)} \\ 3(2kd - l + 1)(d - 1) &> (d-1)(6kd - d + 2 - 3l) \\ 6kd - 3l + 3 &> 6kd - d + 2 - 3l \\ d &> -1 \end{aligned}$$

Since $d \geq 1$, we have that the $\mathsf{SFR}(\mathcal{OIP})$ and the $\mathsf{AFR}(\mathcal{OIP})$ for $l > 1$ is smaller than for $l = 1$, thus $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$. □