Indexing Temporal Relations for Range-Duration Queries

Matteo Ceccarello
University of Padova
Padova, Italy
matteo.ceccarello@unipd.it

Anton Dignös, Johann Gamper
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy
{dignoes,gamper}@inf.unibz.it

Christina Khnaisser
Université de Sherbrooke
Sherbrooke, Canada
christina.khnaisser@usherbrooke.ca

ABSTRACT

Temporal information plays a crucial role in many database applications, however support for queries on such data is limited. We present an index structure, termed RD-INDEX, to support *range-duration queries* over interval timestamped relations, which constrain both the *range* of the tuples' positions on the timeline and their *duration*. RD-INDEX is a grid structure in the two-dimensional space, representing the position on the timeline and the duration of timestamps, respectively. Instead of using a regular grid, we consider the data distribution for the construction of the grid in order to ensure that each grid cell contains approximately the same number of intervals. RD-INDEX features provable bounds on the running time of all the operations, allow for a simple implementation, and supports very predictable query performance. We benchmark our solution on a variety of datasets and query workloads, investigating both the query rate and the behavior of the individual queries. The results show that RD-INDEX performs better than the baselines on range-duration queries, for which it is explicitly designed. Furthermore, it outperforms state of the art indexes also on mixed workloads containing queries that constrain either only the duration or the range along with range-duration queries. Finally, the size of the RD-INDEX is in all settings smaller than the competitors.

CCS CONCEPTS

• **Information systems** → **Temporal data**; **Database query processing**.

ACM Reference Format:
Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2023. Indexing Temporal Relations for Range-Duration Queries. In *35th International Conference on Scientific and Statistical Database Management (SSDBM 2023), July 10–12, 2023, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3603719.3603732

1 INTRODUCTION

Temporal information plays a crucial role in many database applications: in fact, many database management systems and the SQL standard [29] provide automated version control of the data and time travel facilities, allowing to efficiently access past history. Past research mainly concentrated on efficient solutions for important temporal operators, such as temporal aggregation [9, 25, 31, 35], temporal joins [11, 17, 36], and time travel [24] queries. All these approaches consider only the position of intervals along the timeline, ignoring another important aspect, namely the *duration* of intervals. As a result, index structures to support more general selection queries that constrain both the duration and the position in time of intervals have been missing [8, 16, 29] until recently [4]. In many application domains, however, both aspects of temporal information are useful to formulate queries.

Example 1.1. As a concrete use case, consider the use of antibiotics in healthcare. Antibiotic resistance is a world challenge, and the emergence of new resistance factors is very difficult to monitor and to predict due to the diversity of antibiotic usage and events (e.g., environment, species evolution, medical practices, etc.) [21, 30]. Selecting the most appropriate antibiotic and the appropriate treatment duration is an essential step to reduce antibiotic resistance [21]. Thus, defining guidelines for the duration of antibiotic treatments, measuring the adherence to these guidelines, and developing stewardship tools regarding antibiotics usage can help clinicians in choosing the optimal treatment considering the patient's medical history [30, 41]. Such measures should be implemented at a national level in order to monitor and audit antibiotic resistance on a larger scale. In this context, the following types of temporal queries are frequent:

- $Q1$: *"Find all antibiotics prescriptions from October 1, 2016 to March 31, 2017."*
- $Q2$: *"Find all antibiotics prescriptions with a treatment duration between 5 and 8 days."*
- $Q3$: *"Find all antibiotics prescriptions from October 1, 2016 to March 31, 2017, with a treatment duration between 1 and 2 weeks."*

The first query $Q1$ retrieves tuples based on the position of the events on the timeline; we call it *range query*. In contrast, query $Q2$ imposes constraints on the duration of matching events, and we call it *duration query*. Finally, query $Q3$ constrains both types of information; we call it *range-duration query*. This type of queries can be found and have been reported as a primitive operation in other application scenarios that deal with interval data, e.g., in air traffic analysis [5, 39, 40], event detection for video surveillance [34], or the analysis of clinical data [6].

Existing index structures typically support only one of the two aspects, either the position of the interval on the timeline or the duration of the interval. For instance, the well-known relational interval tree [27] is optimized for efficiently determining temporal relationships between intervals but not interval lengths. In the worst case, the entire index tree must be traversed if a query solely contains restrictions on the interval length. Alternatively, the duration of the intervals can be indexed straightforwardly using a classic data structure such as a B-tree. In this case, however, queries constraining only the range of the intervals will need to traverse

SSDBM 2023, July 10–12, 2023, Los Angeles, CA, USA

ACM ISBN 979-8-4007-0746-9/23/07…$15.00
https://doi.org/10.1145/3603719.3603732

the entire tree. Combining the two indexes is typically inefficient since a query can have different selectivities in the two dimensions. Therefore, to efficiently support workloads involving a mix of all three types of queries mentioned above we seek a new index structure that supports both dimensions at the same time.

In this paper, we introduce RD-INDEX, a novel two-dimensional data structure that indexes time intervals both on their position on the timeline and their duration. Our index structure partitions the intervals in a grid according to their start times and durations. Rather than constructing a regular grid, the boundaries between the grid cells are determined by taking into account the data distribution. Such a strategy ensures that each cell contains approximately the same number of intervals, with the exception of some edge cases if the distribution of the intervals is extremely skewed. The uniform distribution of the data over all grid cells allows to obtain very predictable query times, which are proportional to the selectivity of the query. We prove that the time for answering a range-duration query with RD-INDEX is $O(\frac{n}{s^2}\log\frac{n}{s}+\frac{n}{s}+s^2+k)$, where n is the size of the input relation, k is the number of intervals matching the query predicate, and s is the page size. The index can be constructed in $O(n\log n)$ time. The page size s is the only parameter of our index structure, and it is independent of the data distribution. The index structure also lends itself to a rather simple implementation. While being explicitly designed to address range-duration queries, RD-INDEX also supports range-only and duration-only queries efficiently.

We present the results of a detailed experimental evaluation. The results show that the overhead introduced by the data structure is indeed negligible and that the running time in practice is largely proportional to the selectivity of the query. This is in contrast to the competitors we compare to. On range-duration queries we find that RD-INDEX clearly outperforms the competitors. On mixed workloads comprising all three types of queries, we find that RD-INDEX outperforms the competitors in the vast majority of the workloads, even for cases for which specialized solutions exist.

Our contributions can be summarized as follows:

- We describe RD-INDEX, a novel index structure that supports temporal queries involving both the duration and the range of time intervals.
- We provide efficient algorithms for constructing and querying the index.
- We prove bounds on the performance of RD-INDEX, which can be tuned with a single page size parameter s.
- We provide an extensible open source implementation, which we benchmark against state-of-the-art competitors, showing significantly better performance across several workloads.

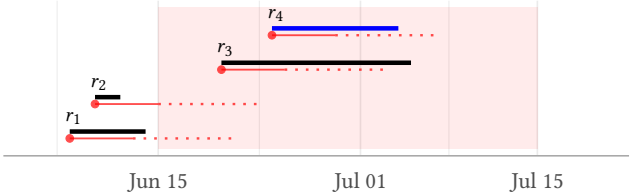
We lay out the fundamental concepts underlying our approach in Section 2, before reviewing the state of the art in Section 3. Our data structure is introduced in Section 4, and the complexity of all operations is analyzed in Section 5. Experimental results are presented in Section 6, before drawing our conclusions in Section 7.

2 PRELIMINARIES

We assume a linearly ordered, discrete time domain, Ω^T. A time interval is a set of contiguous time points, and $t=[t_s,t_e)$ denotes the closed-open interval of points from t_s to t_e. We use $|t|=t_e-t_s$

	drug	T_s	T_e	duration
r_1	Amoxicillin	June 08	June 14	(6 days)
r_2	Amoxicillin	June 10	June 12	(2 days)
r_3	Ceftriaxone	June 20	July 05	(15 days)
r_4	Levofloxacin	June 24	July 04	(10 days)

(a) Sample of relation with antibiotic prescriptions



(b) Range-duration query

Figure 1: Running example.

to denote the duration of time interval t and $t\cap t'$ to denote the set of time points shared by two intervals t and t', which, if not empty, is itself an interval. The schema of a temporal relation is given by $R=(A_1,\ldots,A_m,T)$, where $A_1,\ldots,A_m$ are the non-temporal attributes with domains Ω_i and T is the time interval attribute with domain $\Omega^T\times\Omega^T$ representing, for instance, the tuple's valid time. A temporal relation $\mathbf{r}$ with schema R is a finite set of tuples, where each tuple has a value in the appropriate domain for each attribute in the schema. We use $r.A_i$ to denote the value of attribute A_i in tuple r, and $r.T=[r.T_s,r.T_e)$ to refer to its time interval.

The index we propose efficiently supports the three following types of temporal queries (defined as in [4]).

Definition 2.1 (Range query). Given a temporal interval $t=[t_s,t_e)$ and a temporal relation $\mathbf{r}$, a *range query* is defined as

$$Q(\mathbf{r},t)=\{r\in\mathbf{r}:r.T\cap t\neq\emptyset\}$$

Definition 2.2 (Duration query). Given a duration interval $d=[d_{min},d_{max}]$ and a temporal relation $\mathbf{r}$, a *duration query* is defined as

$$Q(\mathbf{r},d)=\{r\in\mathbf{r}:|r.T|\in[d_{min},d_{max}]\}$$

Definition 2.3 (Range-duration query). Given a temporal interval $t=[t_s,t_e)$, a duration interval $d=[d_{min},d_{max}]$, and a temporal relation $\mathbf{r}$, a *range-duration query* is defined as

$$Q(\mathbf{r},t,d)=\{r\in\mathbf{r}:r.T\cap t\neq\emptyset\wedge|r.T|\in[d_{min},d_{max}]\}$$

A range query retrieves all tuples whose time interval intersects with the query range t. A duration query retrieves all tuples whose time interval has a duration that is between d_{min} and d_{max}. A range-duration query is a combination of the former two.

Example 2.4. As a running example, we consider real-world drug prescriptions from the MIMICIII open source database [22] (cf. use case in Example 1.1). It stores antibiotic prescriptions, characterized by a start date and an end date of the prescription and the duration of the treatment. An excerpt of four tuples is shown in Figure 1a. Consider the following range-duration query: Retrieve all prescriptions in the period from June 15 to July 15 with a treatment duration between 5 and 15 days. Figure **??** shows a graphical representation of the relation and the query, where the time intervals

are drawn by thick solid horizontal lines. The red area indicates the range constraint t. The red segment below of each timestamp interval denotes the duration constraint, where the dotted red line indicates the range $[d_{min}, d_{max}]$. Hence, an interval satisfies the range constraint if it intersects with the red area, and it satisfies the duration constraint if its end point is within the dotted red line. Tuple r_1 satisfies only the duration constraint, tuple r_2 satisfies neither constraint, tuple r_3 satisfies only the range constraint, and tuple r_4 satisfies both.

3 RELATED WORK

The type of selection queries we are studying in this paper are queries with a conjunctive predicate, where one predicate in the conjunction restricts the position of intervals on the time line and the second the duration of intervals. In this section, we review indexing structures that are (partially) suitable for such selection queries, and also review structures similar to our approach that are used for interval joins.

Range-only queries. There are several approaches devoted to indexing interval timestamped data. Edelsbrunner's Interval Tree [19] is one of the most popular indexing structure for intervals. It is asymptotically optimal for selection queries involving the overlap predicate, and there exists an implementation using standard relational database technology based on B-tree indexes [27]. A shortcoming of the interval tree is that, in contrast to our indexing structure, it does not provide a mechanism to restrict the duration of intervals, and thus can only solve one part of a range-duration query. A similar indexing structure is the segment tree [7]. It builds disjoint segments over intervals at the leaf level using all start and end points in a relation, and recursively merges segments in intermediate nodes of the tree. This data structure was originally designed for point queries over intervals (also known as time travel queries), i.e., for retrieving all intervals overlapping a given time point. The segment tree also supports selection queries with the overlap predicate given a query interval, albeit in this case a duplicate elimination step for intervals retrieved multiple times is required. Another index structure that support time travel queries is the timeline index [23, 24]. The timeline index stores the start and end points of intervals in an event list in sorted order and allows to retrieve all tuples that overlap a given time point by scanning through the event list and discarding tuples that ended before the given time point. To avoid scanning through the entire event list, the index maintains regular checkpoints that store all tuples that overlap the time point of the checkpoint. Similarly to the interval tree, the segment tree and timeline index do not support to restrict the duration of intervals.

The HINT index structure [14] also addresses selection queries with the overlap predicate. It is based on a hierarchy of grids of increasing granularity. To efficiently handle skewed data, it uses an additional index of non-empty partitions. HINT is very efficient at indexing the position of intervals on the timeline, but does not index their duration: for range-duration queries this implies that the duration constraint has to be checked using an expensive filtering of the result of the range query.

Range-duration queries. In a two dimensional space, intervals can be represented as 2D points, where one dimension is the start point and the second dimension is either the end point or the duration of an interval. In such a space, a selection query with the overlap predicate corresponds to a selection query over a 2D area. For this, multidimensional indices can be used. R-trees [1, 3] are multidimensional indices that group objects in a multidimensional space using minimum bounding (hyper) rectangles. Quadtrees and octrees [20, 38, 42] recursively divide the space into partitions and place objects into the best fitting partition according to some criteria. All of the aforementioned indexes are linked data structures, which suffer from poor locality, both when implemented in-memory and on disk. In contrast, our RD-INDEX can be implemented by means of simple arrays, and thus enjoys high cache locality.

Another multidimensional index structure is the grid file [33]. In the context of time intervals, the idea would be to partition the span of durations and of starting times into cells of equal width, thus allowing efficient access to both dimensions. The main drawback of this data structure is that in case of skewed data distributions the load of the cells is unequal, which might significantly harm the performance.

A recent approach to multidimensional indexing is that of *learned indexes* [26]: the proposition is that index structures are *models* mapping keys to records. Therefore machine learning models can be used to provide this mapping, in lieu of the classic data structures. In particular, Flood and Tsunami [18, 32] use Recursive Model Index [26] and a variant of decision trees to model the position of records in the database, adapting to the distribution of the data and of the query workload. Our approach shares some ideas with this line of work, namely adapting to the data distribution by means of the conditional cumulative distribution function. However, while our index supports both insertions and deletions, both Flood and Tsunami are tailored at read-only workloads. Furthermore, we prove bounds on the worst case running times for all the operations, while [18, 32] provide an empirical evaluation. Finally, our approach is arguably simpler, in that it is based just on sorting and iterating through records.

Very recently, Behrend et al. [4] proposed an index, named PERIOD-INDEX★, that explicitly supports range-duration queries. The index partitions the time domain in *buckets*. An interval is assigned to all buckets it intersects with. Within each bucket, intervals are further partitioned in *levels* according to their duration, with the minimum duration indexed within each level decreasing geometrically. To support efficient indexing along the start time dimension, each level is further partitioned in the time domain. This data structure is adaptive to the distribution of start times, while it assumes a Zipf-like distribution for the duration of the intervals. Our index structure removes this assumption, thus supporting datasets with arbitrary distributions of the tuples' duration. Furthermore, our index features only one data-independent parameter, instead of the two data-dependent parameters of PERIOD-INDEX★, and it allows to control whether to index first by duration or time. Moreover, we do not replicate intervals in the index, yielding a significantly smaller structure, thereby avoiding the consequent possible performance degradation. Finally, PERIOD-INDEX★ does not support updates of the index.

Other types of queries. In recent years, algorithms for interval joins, which can be seen as a sequence of range queries, have been

actively studied. Approaches based on the timeline index [24] process sets of intervals as sorted event lists of their start and end points. The interval join is computed by scanning these event lists in an interleaved fashion, thereby keeping and joining sets of active intervals, i.e., intervals whose start has been encountered but not the end point. To improve the performance of the original linked list data structure for storing active intervals, a gapless hash map has been proposed in [36] that provides a higher performance for scanning active intervals. The same idea has also been extended for joins using general Allen's predicates [37] rather than only overlap predicates. The works in [10, 11, 17] compute an interval join using sorting and backtracking. First, the input relations are sorted by start time and then an interleaving merge-join is performed to compute the temporal join between the two relations. Other approaches [12, 15] for the interval join are based on partitioning intervals according to their position and then produce the join result by joining relevant partitions. While all these approaches for the interval join provide mechanisms to join overlapping intervals, in contrast to our work they are not applicable for general selection queries as they always require to read the entire relations. Moreover, the duration of intervals is not considered at all in these works.

4 THE RD-INDEX STRUCTURE

4.1 Overview

An interval can be completely described by its starting time and duration, alternatively, to its starting and ending time. Therefore, a set of temporal intervals can be represented as a set of points in a two dimensional space, with one coordinate being the start time and the other the duration. A range-duration query $Q(\mathbf{r}, t, d)$ with $t = [t_s, t_e)$ and $d = [d_{min}, d_{max}]$ in such a space is represented by a polygon containing all tuples $r \in \mathbf{r}$ such that

$$|r.T| \in [d_{min}, d_{max}] \tag{1}$$
$$r.T_e > t_s \tag{2}$$
$$r.T_s < t_e \tag{3}$$

An example of this representation of intervals and queries is shown in Figure 3, along with the partitioning of this space induced by our index.

The RD-index we are presenting is a two-dimensional grid, partitioning the tuples into disjoint buckets according to the start time and the duration of the intervals. The boundaries between cells are defined by using the empirical cumulative distribution function of the tuples' duration and the starting times, so that each cell contains approximately the same number of intervals, which corresponds to the *page size* s of our index. This allows the index to adapt to the distribution of the input and to different scenarios. In a main memory scenario, the parameter s could be set such that a cell fits in a cache line. In an external memory setting, it might be set to the disk block size.

To simplify the presentation, in the following we will focus on the timestamp interval attribute T of relation $\mathbf{r}$: in the discussion we assume that each interval being inserted in the index is associated with a reference to the tuple in $\mathbf{r}$ it belongs to.

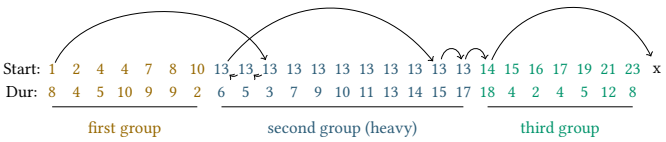


Figure 2: Partitioning a sequence of intervals sorted by start time using NextSubseq with parameter $b = 9$, with the start time being the *key function*.

4.2 Index Construction

The grid structure of the RD-index partitions an array of tuples first along either the start time or duration dimension of the intervals, and then along the other. The choice of which dimension to index first may impact the performance of the index, depending on the query workload and the data distribution (cf. Section 6). In the following we assume that the start time dimension is partitioned first, followed by the duration dimension. All the descriptions, considerations, and proofs also hold with the dimensions swapped.

Before describing the algorithm to build the index, we present the subroutine NextSubseq, which partitions an array of tuples that is sorted according to a given key function. We will use this subroutine to determine columns and cells of the grid structure, using first the start time and then duration as keys. Given an index h and a size parameter b, NextSubseq returns a subsequence starting at h that either contains at most b tuples, or contains tuples that all share the same key. Additionally, all the tuples with the same key are part of the same subsequence. The pseudocode is reported in Algorithm 1 and works as follows. Starting from index position h, if there are fewer than b elements after h then we return all the tuples from h onwards. Otherwise, we look at the tuple at position $h' = h + b$ and consider two cases[1]. If the tuples at position h and h' have the same key, then we scan forward until the first tuple with a different key occurs (lines 4–5). Otherwise, we scan backward until two consecutive tuples have different keys (lines 6–7). In both cases, the rationale is to avoid splitting runs of same-key tuples between different subsequences.

Note that Algorithm 1 might return a subsequence with more than b elements if and only if all share the same key. In such case we deem the returned subsequence *heavy*, otherwise we deem it *light*. As we shall see in Section 4.3, *heavy* subsequences are easy to deal with for our index.

Example 4.1. Figure 2 depicts three invocations of NextSubseq on a sequence of sorted start times, with parameter $b = 9$. The first jump by 9 positions would split the run of intervals with start time 13. Therefore, the algorithm iterates back until the first start time < 13. The second invocation would again split the same run since it contains more than 9 intervals with start time 13. This time, since the endpoints of the jump have the same value, the algorithm iterates forward until the last interval with the same start time, thus finding a *heavy* subsequence. The last jump defines the third group.

We are now ready to describe the index construction procedure BuildIndex, which is shown in Algorithm 2. Let s be the *page size*

[1]Therefore h' is the end index of the subsequence, *non-inclusive*.

Algorithm 1: NextSubseq $(\mathbf{r}, h, b, k)$

Input: Relation $\mathbf{r}$ sorted according to k; current index position h; subsequence size b; *key* function k
Output: Subsequence of $\mathbf{r}$ starting from position h, either of size $\leq b$ or with all tuples having the same key

```
1  if h + b ≥ |r| then
2      return subsequence ⟨r_h, ..., r_{|r|-1}⟩;
3  h' ← h + b;
4  if k(r_h) = k(r_{h'}) then
5      while h' < |r| ∧ k(r_h) = k(r_{h'}) do h' ← h' + 1;
6  else
7      while k(r_{h'-1}) = k(r_{h'}) do h' ← h' − 1;
8  return subsequence ⟨r_h, r_{h+1}, ..., r_{h'-1}⟩;
```

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

Input: Temporal relation $\mathbf{r}$ and page size parameter s
Output: Grid G partitioning $\mathbf{r}$ by start time and duration, along with auxiliary arrays.

```
1  grid ← [][];
2  col_minstart ← [];
3  col_maxend ← [];
4  cell_mindur ← [][];
5  cell_maxdur ← [][];
6  Sort r by interval start time;
7  h ← 0 /* position in r                                   */
8  i ← 0 /* column index                                    */
9  while h < |r| do
10     column ← NextSubseq(r, h, s^2, r.T → r.T_s);
11     col_minstart[i] ← min{r.T_s : r ∈ column};
12     col_maxend[i] ←
         max{col_maxend[i-1], {r.T_e : r ∈ column}} ;
13     Sort column by duration;
14     k ← 0 /* position in column                          */
15     j ← 0 /* cell index                                  */
16     while k < |column| do
17         cell ← NextSubseq(column, k, s, r.T → |r.T|);
18         Sort cell by end time;
19         grid[i][j] ← cell;
20         cell_mindur[i][j] ←
             min{|r.T| : r ∈ cell};
21         cell_maxdur[i][j] ←
             max{|r.T| : r ∈ cell};
22         j ← j + 1;
23         k ← k + |cell|;
24     i ← i + 1;
25     h ← h + |column|;
26 return (grid, col_minstart, col_maxend, cell_mindur,
    cell_maxdur);
```

parameter, and $\mathbf{r}$ be the relation to be indexed. First, we sort $\mathbf{r}$ by increasing start time. Then, we repeatedly invoke NextSubseq (line 10) to divide the relation into *columns* of s^2 tuples each, based on their start times. Defining columns in this way allows, intuitively,

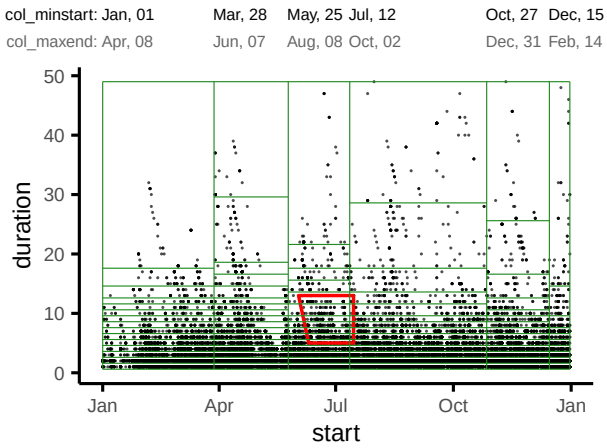


Figure 3: Instantiation of RD-index with page size $s = 70$ on the example dataset. Above the plot we report the `col_minstart` and `col_maxend` arrays.

to then further partition each column in s cells of s elements each As a special case, if there are more than s^2 tuples with the same start time, NextSubseq will assign them to the same column, which we then deem *heavy*. We also keep track of the minimum start time in each column using an auxiliary array `col_minstart`. Similarly, the array `col_maxend` stores the cumulative maximum end time in the columns, i.e., the maximum end time found so far in the relation. Both of these arrays will be used at query time: the first to find the first column to inspect for a given query, the second to determine when to stop iterating through columns. Note that `col_maxend` records the *cumulative* maximum end time of columns. This ensures that no interval in columns $\leq i$ ends after `col_maxend[i]`, which will be useful at query time.

Each column is further partitioned in *cells* in a similar way. First we sort the tuples in the column by increasing duration. Then we define cells of size s by repeatedly invoking NextSubseq (line 17). Again, if there are more than s tuples with the same duration, they will all be assigned to the same cell, which will then be deemed *heavy*. Similarly to columns, also cells are complemented by two arrays of ancillary information: `cell_mindur` stores the minimum duration in each cell to be used at query time to find the first cell to inspect, while `cell_maxdur` stores the maximum duration in each cell, which at query time will determine when to stop iterating through cells.

Finally, tuples in each cell are sorted by the end time. This is useful at query time, since it allows to stop queries early, as we shall discuss in the proof of Theorem 5.3.

We now formally define light and heavy columns and cells, since they play a key role in the proof of the performance of our index structure.

Definition 4.2. For a given page size s, a *heavy* column (resp. cell) contains $> s^2$ (resp. $> s$) intervals. Conversely, a *light* column (resp. cell) contains $\leq s^2$ (resp. $\leq s$) intervals.

Example 4.3. Figure 3 shows the grid (in green) constructed by Algorithm 2 on our example relation from Figure 3, with page

size parameter $s = 70$. Note that the columns, which contain $s^2 = 4\,900$ tuples each, span different ranges of start times, adapting to the density of the points. Within each column, the points are partitioned according to the distribution of durations. Since many drug prescriptions have the same short durations, the cells at the bottom of the columns are *heavy* (or span only a few different duration values). In this setting, a uniform grid would be heavily imbalanced. The red box denotes the area corresponding to the query of Example 2.4.

4.3 Querying the Index

Given a range-duration query with time range $t = [t_s, t_e)$ and duration interval $d = [d_{min}, d_{max}]$, recall that a tuple $r \in \mathbf{r}$ satisfies the query if the conditions (1), (2), and (3) specified in Section 4.1 are met.

The pseudocode for querying the index structure is reported in Algorithm 3. First, we seek the index of the last column that might contain matching tuples. To this end, we perform a binary search on the array `col_minstart` to find the last column i such that the minimum start time in the column (which is the column bound) is strictly less than the query end time t_e (line 2). This ensures that the column contains at least one tuple satisfying condition (3). Then, we iterate *backwards* through columns until column i cannot possibly contain tuples satisfying the query. For this, we use the support array `col_maxend`: if `col_maxend[i]` $\leq t_s$ then we know that all the columns at index $\leq i$ contain tuples that stop earlier than the start of the query range. Hence, we can avoid inspecting them because of condition (2).

For each column that we consider, a binary search on the array `cell_mindur[i]` is performed, looking for the last cell j such that the minimum duration in the cell (which is the cell bound) is $\leq$ to the maximum duration d_{max} specified in the query (line 4). Doing so ensures that at least one tuple in the cell satisfies the upper bound of condition (1). Then, we iterate backwards through the cells until we reach a cell whose maximum duration is less than the minimum duration d_{min} of the query. At this point we stop since condition (1) can no longer be satisfied.

Finally, we iterate through the tuples of each considered cell by decreasing end time and stop as soon as condition (2) is no longer satisfied (line 7). All the intervals that satisfy the query are returned in the result.

4.4 Updating the Index

Our index data structure can be extended to support both insertion and removal of tuples. In any case, we stress that our focus is on the analysis and exploration of data rather than on updates.

Interval Insertion. To insert a tuple r, we query the index for the start time $r.T_s$and the duration $|r.T|$ to identify the cell that should contain r. Inserting new intervals into cells might make them grow too large to be able to maintain the performance guarantees on the query time. Luckily, as we shall see in Section 5, *heavy* columns and cells do not present issues upon insertions by virtue of containing intervals that all share either the same start time or the same duration. If a light column exceeds size s^2, we replace it with two new columns. Similarly, if a light cell exceeds size s, we replace it with two new cells.

Algorithm 3: QUERY (`grid`, $[t_s, t_e)$, $[d_{min}, d_{max}]$)

Input: A range duration query with time range $[t_s, t_e)$ and duration range $[d_{min}, d_{max}]$; An index `grid` with the ancillary arrays `col_minstart`, `cell_mindur`, `col_maxend`, and `cell_maxdur`

```
1  res ← ∅;
2  i ← argmax_i col_minstart[i] < t_e;
3  while i ≥ 0 ∧ t_s < col_maxend[i] do
4      j ← argmax_j cell_mindur[i][j] ≤ d_max;
5      while j ≥ 0 ∧ cell_maxdur[i][j] ≥ d_min do
6          for r ∈ grid[i][j] do
7              if r.T_e ≤ t_s then break;
8              if |r.T| ∈ [d_min, d_max] ∧ r.T_s < t_e then
9                  res ← res ∪ {r};
10         j ← j − 1;
11     i ← i − 1;
12 return res
```

Splitting a column entails to consider all the intervals it contains, using NEXTSUBSEQ with $b = s^2/2+1$ to find the breakpoint at which to split (this way, we balance the size of the new columns). For each of the two new columns that replace the original column, we apply NEXTSUBSEQ to split them into cells, exactly as in the inner loop of the index construction. The array `col_minstart` and `col_maxend` are updated to reflect the replacement of the old column with the new ones.

Similarly, to split a cell that exceeds size s in column i we use NEXTSUBSEQ with $b = s/2 + 1$ to find a new breakpoint and replace the cell with two new cells. The auxiliary structures `cell_mindur[i]` and `cell_maxdur[i]` are updated accordingly.

Both in the case of column and cell splitting, we sort all the intervals in the newly created cells by end time.

Interval Removal. As for the removal of a tuple r from the index, we query the index to find the cell `grid[i][j]` that contains $r.T$ and remove the interval from the cell. As a consequence, the cell might contain fewer than $s/2$ items. As we shall see with Lemma 5.2, it is crucial for the performance of the index that cells contain at least $s/2$ intervals.

Therefore, upon removal of an element from a cell, we check whether the sum of elements of the cell and either of the adjacent ones is less than s. In such case, we *merge* the two cells, i.e., we replace them with a single cell where all intervals are then sorted by decreasing end time. After cells are merged, the arrays `cell_mindur[i]` and `cell_maxdur[i]` are updated as well to reflect the changes.

Similarly, a removal might cause a column to have fewer than $s^2/2$ elements. We then apply a similar reasoning. If the sum of the number of items in the column and either adjacent ones is smaller than $s^2/2$, we merge the two columns, i.e., the two columns are replaced by a single one. NEXTSUBSEQ is then called to find the breakpoints to divide the newly created column into cells. After the two columns are merged, the arrays `col_minstart` and `col_maxend` are updated to reflect the changes.

5 ANALYSIS

In this section, we provide guarantees on the time required by all the operations supported by our index structure. We assume that the index is built by partitioning first in the time dimension, and then in the duration dimension. The asymptotic results presented in this section hold for both dimension orderings.

5.1 Querying the Index

The proofs of the following lemmas and theorem are given in the full version [13] for the sake of space.

Lemma 5.1. *A heavy cell in a heavy column contains only copies of the same interval.*

Lemma 5.2. *The RD-index with parameter s over n intervals has $O\left(\frac{n}{s^2}\right)$ columns, each having $O\left(\frac{n}{s}\right)$ cells.*

Theorem 5.3. *Given an index over a set of n intervals and a page size s, the time for answering a range-duration query is*

$$O\left(\frac{n}{s^2}\log\frac{n}{s}+\frac{n}{s}+s^2+k\right)$$

where k is the number of intervals matching the query predicate.

The above theorem exposes a fundamental tradeoff of our data structure: using a smaller page size allows to improve the precision of the data structure (by looking at fewer intervals that are not part of the query output), while at the same time increasing the number of columns that need to be queried. In Section 6.4 we investigate experimentally the effect of s on the performance, finding that the best performance is attained for $n/s^2 \in [50, 500]$.

The intuition in the proof of Theorem 5.3 is that each cell visited by a query is either *light*, and thus contributes at most $O(s)$ time to the running time, or is *heavy*, in which case all its intervals are part of the output, contributing $O(k)$ to the running time.

The choice of the order of partitioning has no impact on the theoretical complexity results, however it affects the practical performance as we will discuss in Section 6 (Figure 7). It turns out that generally it is better to index first the duration and then the start time, as summarized in the following observation.

Observation 1. *Changing the order in which dimensions are indexed does not change the asymptotic behavior of RD-index, but might affect the practical performance. A duration query is insensitive to the position of start times. Conversely, a range query can benefit from a partition of the end times, which is implied by the duration partitioning. Hence, indexing first by duration might prove beneficial.*

5.2 Index Construction and Update

For the sake of space, the proofs of the following theorems can be found in the extended version [13].

Theorem 5.4. *Given a set of n intervals, building the index requires time $O(n \log n)$.*

Theorem 5.5. *Inserting a tuple r into the index requires time $O\left(\log n/s + n/s + s^2 \log s\right)$.*

Theorem 5.6. *Removing a tuple r from the index requires time $O(\log n/s + n/s)$.*

6 EXPERIMENTAL EVALUATION

To frame our evaluation, we consider the following data structures as baselines. The implementation of B-Tree provided by the Rust standard library, which is optimized for CPU cache usage; intervals are indexed by duration in this case. The Interval-Tree index [27], which we implemented ourselves, indexing intervals by their position on the timeline. The Grid-File [33] and Period-Index★ [4], which we also implemented, and which index both start times and durations. We also consider the R-Tree as a baseline, specifically the R*-Tree[2] [3]: intervals are mapped to points identified by the start time and duration of the interval, and then indexed by the R*-Tree. Furhtermore, we consider HINT [14], extending the original C++ implementation[3]to support range-duration queries. Our proposed data structure will be denoted with RD-index-td when start time is indexed before duration, and with RD-index-dt otherwise. In cases where the order of the dimensions is not relevant to the discussion, we will use RD-index instead. To account for the potential shortcomings of our implementations, we will also evaluate the relative performance of the data structures with implementation-independent metrics [28].

Setup and Datasets. We implemented our index and the baseline competitors in Rust 1.44.1, using a configurable and extensible framework [2]. Code and data are available at https://github.com/Cecca/temporal-index. The experiments presented in this section were run on a machine equipped with 94 GB of memory and a Intel®Xeon®CPU E5-2667 v3 @ 3.20GHz processor.

As a benchmark we consider the following datasets and workloads. `Flight`: A set of 701 353 flights, identified by their takeoff and landing time at the granularity of minute, covering August 2018. Query ranges on this dataset are generated at random. Time range durations are uniformly distributed between one and 31 days, and duration ranges are uniformly distributed between one minute and one day. `Webkit`: 1 547 419 file edits in the Webkit source code repository. Intervals represent the timespan between successive edits to a file. Query ranges on this dataset are generated at random. Time range durations are uniformly distributed between one minute and one year, and duration ranges are uniformly distributed between up to three years. `MimicIII`: 4 134 909 drug prescriptions from the open MimicIII database [22]. Each prescription is characterized by its start and end day. Queries, generated at random, span the entire domain of times and durations: the former take values $\in [1, 40\,251]$, the latter $\in [1, 200]$. The large span of start times (110 years) is due to the anonymization procedure applied to the database. `Synthetic`: Randomly generated datasets with 10 million intervals by default. The interval start times are uniformly distributed in $[1, n]$, where n is the size of the dataset; interval durations follow a Zipf distribution with $\beta = 1$.

6.1 Robustness of Index Structures Across Different Workloads

In the first set of experiments, we consider both real-world and synthetic datasets (with 10 million intervals). Table 1 reports, under different combinations of dataset/query workload, an overview on

[2]https://github.com/georust/rstar
[3]https://github.com/pbour/hint

Table 1: Performance of indices on different workloads: queries per second (best in blue), index build time (best underlined), index size as number of bytes per interval (best in bold).

		Queries per second | Index build time | Bytes per interval							
dataset	query	RD-index-td	RD-index-dt	Grid-File	Period-Index★	R*-Tree	HINT	Interval-Tree	B-Tree
Synthetic	range-only	415 | 1 024 | **16.1**	602 | 889 | 16.1	468 | 445 | 27.7	242 | 2 392 | 57.1	12 | 3 358 | 80.0	164 867 | 2 848 | 46.6	272 | 3 636 | 50.1	8 | 2 505 | 31.4
	duration-only	842 | 1 030 | **16.1**	734 | 785 | 17.1	77 | 983 | 24.1	69 | 6 139 | 178.1	106 | 3 398 | 80.0	6 | 2 102 | 29.3	12 | 3 636 | 50.1	43 103 | 2 469 | 31.4
	range-duration	11 737 | 1 052 | **16.1**	14 085 | 829 | 16.5	1 581 | 857 | 24.1	1 403 | 6 566 | 178.1	2 591 | 3 402 | 80.0	72 | 2 109 | 29.3	215 | 3 674 | 50.1	502 | 2 529 | 31.4
Flight	range-only	54 945 | 42 | 17.1	57 471 | 44 | **16.1**	49 505 | 20 | 23.2	15 015 | 80 | 40.0	4 921 | 195 | 97.9	1 342 098 | 116 | 52.7	51 813 | 139 | 44.5	2 798 | 39 | 23.3
	duration-only	4 182 | 44 | **16.1**	4 218 | 33 | 21.3	3 791 | 20 | 23.2	538 | 111 | 62.9	378 | 205 | 97.9	169 | 123 | 52.6	683 | 150 | 44.5	714 286 | 43 | 23.3
	range-duration	232 558 | 45 | 16.7	208 333 | 39 | **16.6**	188 679 | 20 | 23.2	29 940 | 95 | 62.9	21 786 | 193 | 97.9	11 439 | 161 | 62.2	43 860 | 139 | 44.5	13 514 | 47 | 23.3
Webkit	range-only	416 | 116 | **16.1**	467 | 115 | 16.1	445 | 56 | 23.0	83 | 540 | 119.0	30 | 468 | 97.7	525 897 | 438 | 76.6	384 | 274 | 48.3	47 | 252 | 36.4
	duration-only	2 544 | 112 | **16.1**	2 520 | 104 | 16.9	1 938 | 52 | 23.0	57 | 559 | 124.4	234 | 469 | 97.7	64 | 264 | 54.6	157 | 274 | 48.3	3 393 | 255 | 36.4
	range-duration	3 159 | 112 | **16.1**	3 133 | 99 | 16.9	2 322 | 51 | 23.0	80 | 737 | 322.6	258 | 468 | 97.7	95 | 263 | 54.6	256 | 274 | 48.3	758 | 252 | 36.4
MimicIII	range-only	372 | 240 | **16.0**	391 | 213 | 16.0	391 | 143 | 23.5	381 | 451 | 24.3	38 | 1 112 | 81.2	298 566 | 596 | 31.5	347 | 717 | 46.6	127 | 138 | 21.2
	duration-only	3 560 | 275 | **16.0**	2 876 | 202 | 17.5	2 075 | 184 | 27.7	520 | 718 | 41.3	452 | 1 175 | 81.2	25 | 580 | 24.1	76 | 733 | 46.6	2 500 000 | 141 | 21.2
	range-duration	10 246 | 241 | **16.0**	10 730 | 201 | 17.5	5 227 | 185 | 27.7	1 779 | 720 | 41.3	1 384 | 1 081 | 81.2	84 | 587 | 24.2	232 | 721 | 46.6	3 347 | 135 | 21.2

the performance of different index structures on three indicators: the queries per second, the time to build the index, and the size of the index. The latter is measured in terms of bytes per interval, i.e., the number of bytes that the index uses for each input interval. Since we are representing intervals as pairs of 64-bits unsigned integers, 16 bytes per interval are required just to represent the data, and thus are a lower bound on this performance metric. Dark blue and light blue cells denote, respectively, the best and second-best performing data structures in terms of queries per second. For index structures that take parameters, we report on the best configuration. In particular, we defer the discussion of the effect of different parameterizations of RD-index to Section 6.4.

We remark that the overall difference in throughput for different workloads is due to the different output sizes: duration-only queries are in general less selective than range-duration queries, hence it takes more time to iterate through the output. This explains why, in general, range-duration queries enjoy a higher throughput across all the index structures.

In terms of throughput, we observe that RD-index always performs better than competitors on range-duration queries. For duration-only queries it is always the second best solution after the B-Tree, while for range-only queries it is always the second after HINT. We will, however, see in the next section how RD-index surpasses both as soon as a few range-duration queries are introduced in the workload. In particular, HINT's implementation stores identifiers of the original records in the index, as opposed to our implementation where the index stores the intervals themselves. The consequence is that HINT suffers from cache misses whenever a duration constraint has to be checked.

We also observe that the grid-file ranks second or third in several cases. Recall that this data structure is rather similar to RD-index, the difference being that the latter is adaptive to the input distribution. This shows the performance benefits of a data structure that takes into account the data distribution.

Concerning the other performance indicators, we note that the index construction time of RD-index is comparable with the one of the Grid-File and B-Tree, and generally much faster than the other approaches. As for the size of the index, RD-index always produces the smallest index, across all tested configurations, using just slightly more than the minimum 16 bytes to represent an interval. The other approaches, especially pointer-based data structures such as B-Tree, Interval-Tree, and R*-Tree require significantly more space. The Period-Index★ has a much higher space requirement compared to the others, since each interval may be replicated several times.

In summary, RD-index is a data structure that provides fast query times, is fast to build, and has negligible space overhead.

6.2 Mixed Query Workloads

We now consider mixed query workloads, consisting of a mix of range-only, duration-only, and range-duration queries. Rather than fixing a particular combination of queries, we use the data of Table 1 to estimate the throughput of workloads composed by any combination of queries. Let f_{rd}, f_r, and f_d be, respectively, the fraction of range-duration, range-only, and duration-only queries in the mixed query workload, with $f_{rd} + f_r + f_d = 1$. Similarly, for a given algorithm and dataset, let ϕ_{rd}, ϕ_r, and ϕ_d be the throughputs of range-duration, range-only, and duration-only queries, as reported in Table 1. As is customary with rates, we use the *harmonic mean* to compute the overall average rate of a mixed workload starting from the rates reported in Table 1: $(f_{rd}/\phi_{rd} + f_d/\phi_d + f_r/\phi_r)^{-1}$.

Figure 4 provides a summary of the best performing algorithm for any workload that can be concocted with the formula above. In each ternary plot, each point in the triangle identifies a combination of range-only, duration-only, and range-duration queries. For instance, the center of each triangle corresponds to a workload composed in equal parts by the three types of queries. Portions of the triangles are colored according to the best-performing index for the corresponding workloads. We observe that RD-index is the

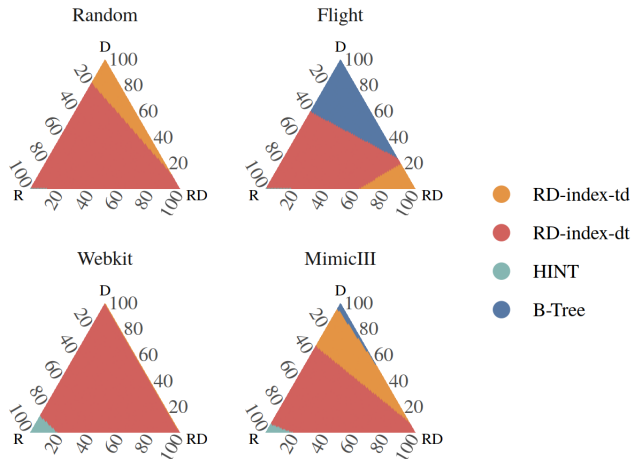


Figure 4: Ternary plots of best performing index structure for different mixed workloads, on the four different datasets. The online supplementary material provides an interactive version of this plot.

best performing index structure in the vast majority of workloads. The exception is for workloads where duration-only queries are the majority (top corner), where the B-Tree outperforms RD-index and workloads with mostly range-only queries (lower-left corner), where HINT is the best index. On mixed workloads, we observe that the performance of RD-index is rather robust to the ordering of indexing dimensions.

The online supplemental material[4] provides an interactive tool to explore the performance of all different index structures on any mixed workload.

6.3 Distribution of Query Times Against Selectivity

We investigate the relationship between the selectivity of queries (i.e., the fraction of the input that satisfies them) and the time taken by different data structures to answer them. Given that range-duration queries constrain both the time and the duration ranges, for a query we can define the *time selectivity* as the fraction of the input that satisfies the time range constraint of the query, and the *duration selectivity* as the fraction of the input satisfying the duration constraint of the query.

For a given dataset, we build a query set such that queries are uniformly distributed on the *time* × *duration* selectivity plane: this way we have queries that are very selective in only one dimension, very selective in both dimensions, or not selective at all. The goal is to investigate the behavior of each index structure for queries with different characteristics. To account for the overhead of measuring time and to level out the effect of the CPU cache, we run each query 100 consecutive times and report the average. For RD-index, we set the page size to $s = 200$, which is a good parameter choice of all the datasets we consider.

Figure 5 reports the results for such a setup, with 1024 queries arranged in a 32×32 grid, running on a synthetic dataset with 10 million intervals, with uniformly distributed start times and Zipf distributed durations. Colors encode the fraction of the dataset inspected by each query. This metric allows a more implementation-independent assessment of the relative performance of different data structures. When reading Figure 5, remember that less selective queries require more time just to iterate through the output. Interestingly, different data structures exhibit different patterns in this plot, as a result of how they access data.

The Interval-Tree and HINT plots exhibit vertical bands. The data structures are able to select intervals only based on their position on the timeline. Therefore, for a fixed time selectivity of the query the fraction of intervals inspected (and thus the time to answer the query) does not depend on the selectivity in the duration dimension, since all the candidate intervals need to be examined. For similar reasons, the B-Tree shows horizontal bands. Data structures that explicitly index both dimensions, instead, tend to exhibit a more *diagonal* pattern, in particular RD-index, with a milder effect for Grid-File and R*-Tree. The pattern exhibited by Period-Index★ tends to be more similar to the Interval-Tree. This means that this index is more responsive to queries that are selective in the time dimension. This is to be expected since Period-Index★ is adaptive to the distribution of start times in the dataset. Its worse performance compared to RD-index and Grid-File for queries of a given selectivity is explained by the fact that some intervals might be represented multiple times in the index.

Figure 6 reports the performance against the *overall* selectivity of the same queries. The performance is assessed in terms of the fraction of the input inspected by each query, which are represented as dots whose position along the x axis encodes their overall selectivity. Ideally, a data structure should answer queries by inspecting just the tuples which are part of the output. This behavior is represented by the black diagonal in Figure 6. We observe that RD-index indeed shows the ideal behavior. As for the Grid-File, since the efficiency in answering a query depends on the density of the cells being considered, the performance is in many cases far from ideal. This can be seen from the fact that several queries are far away from the ideal diagonal. The B-Tree indexes intervals by their duration. As such, duration-only queries are answered most efficiently: In Figure 6 such queries lie on the ideal diagonal. On the other hand, range-only queries are answered by simply enumerating the entire dataset, thus scoring 1 on Figure 6. Similar considerations hold for the Interval-Tree and HINT, with range-only queries performing the best and duration-only queries performing the worst. Finally, Period-Index★ inspects the same tuples multiple times for the majority of queries, which are thus very far away from the ideal diagonal line in the plot.

6.4 Influence of parameters

The aim of this section is to investigate the influence of the page size and the order in which the two dimensions are indexed. We use two datasets with 10 million intervals each: the first has uniform start times and skewed durations (Zipf distribution), the second has skewed start times and uniformly distributed durations. As for the query workload, batches of 10 000 range-duration, range-only, and duration-only queries are considered. The page size is varied between 1 and 10 000.

[4] https://cecca.github.io/temporal-index/

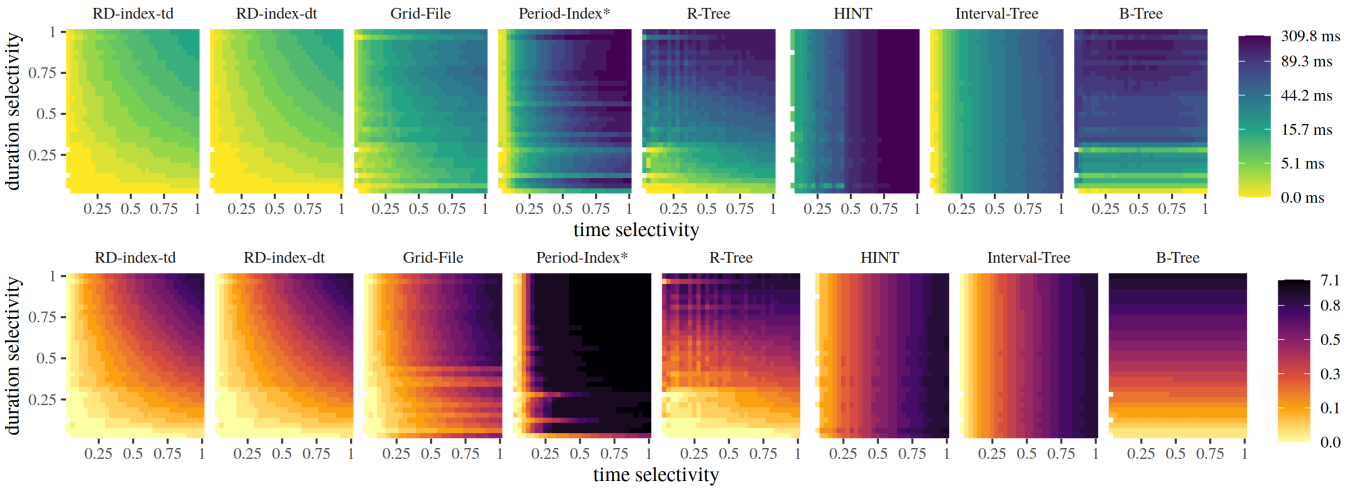


Figure 5: Heatmaps of the performance of index structures against selectivity in the time and duration dimensions. The top row reports the time, in milliseconds, required to answer a query with a given time and duration selectivity. The plots report the fraction of intervals (over the total $n = 10^7$) examined by each query. A fraction larger than 1 means that a query examined the same interval more than once. For readability the color scales are binned every 5 percentiles, therefore the scales are non linear. In both plots, lighter is better.

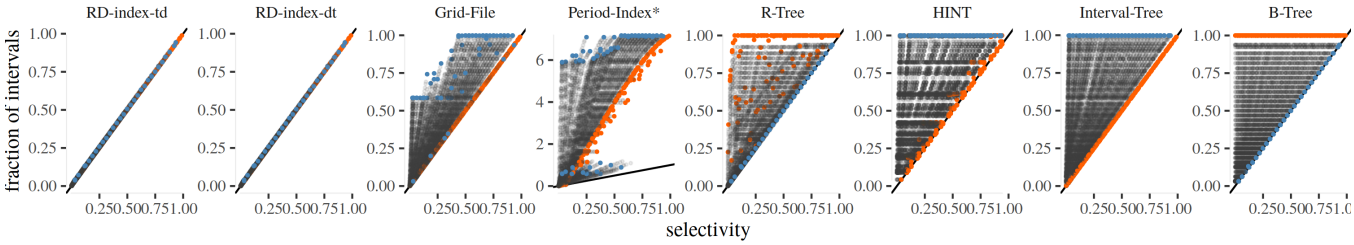


Figure 6: Number of examined intervals against selectivity of the query. Each point represents a query: red dots • are range-only queries, blue dots • are duration-only queries, black dots • are range-duration queries. The black line represents the ideal behavior, in which only matching intervals are examined. Note that all the plots except for the one related to the PERIOD-INDEX⋆ share the y axis: in all plots the black diagonal has slope 1.

Due to the size of the dataset, any page size s above $\sqrt{n} \approx 3\,162$ results in a degenerate configuration, where there is a single column (of size $s^2 = n$) that contains all the intervals. In this case the intervals are partitioned according to a single dimension, with the number of cells controlled by the page size parameter. This situation might also occur at lower values of the page size parameter, depending on the number of distinct values in the dimension being partitioned. On the other hand, for page size 1, each cell of the grid contains only intervals with the same start time and duration, and it contains all of them. Querying the data structure in this case amounts to perform binary searches directly on the values of the domains of start times and durations.

Figure 7 reports the results of this experiment in terms of queries per second. To ease the comparison between the plots, we rescale the throughput by the highest value for each combination of dataset and query workload. Degenerate configurations resulting in a single column are reported as triangles rather than dots. We observe very different trends for different query workloads.

Consider first the dataset with skewed durations. For range-duration queries, both orderings of dimensions exhibit a similar behavior. The peak performance is reached by intermediate values around the page size. If we consider duration-only queries the profile changes. Indexing first by duration (blue line) slightly favors smaller page sizes, which imply smaller columns and thus a more fine grained access to the data. Indexing first by time, instead, requires a duration-only query to traverse all the columns. In this scenario high page sizes are favored, since they translate into fewer columns to be iterated through. For range-only queries we observe a symmetric pattern.

When data has skew on the start times, the patterns exhibited by the two indexing orders are rather different. First, we note that the gap between the best configurations of the two indexing orders for range-duration queries is much wider than with the other dataset. Second, while indexing first by duration exhibits a similar pattern on both datasets, indexing first by start time performs best in the degenerate cases of a single column, i.e., with no partitioning of

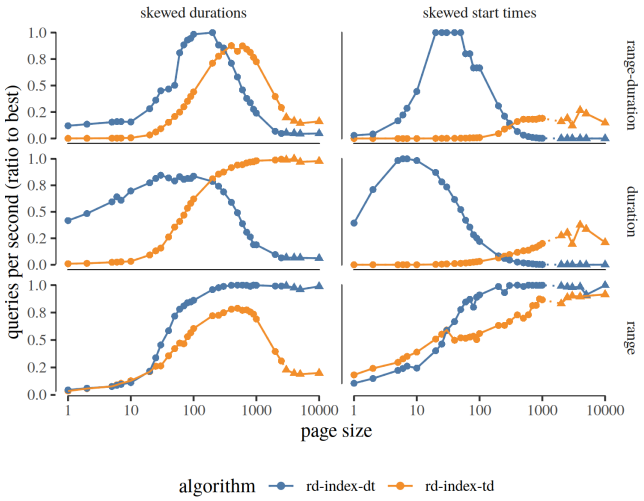


Figure 7: Dependency of the performance, in terms of queries per second, on the page size parameter.

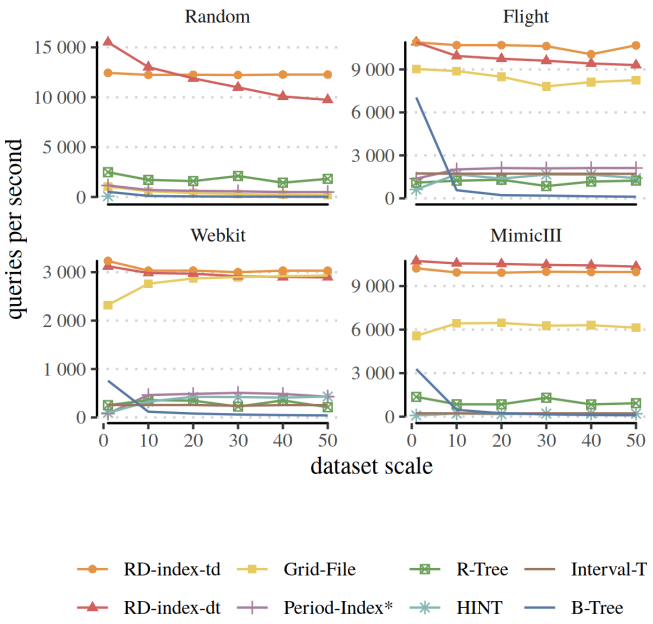


Figure 8: Scalability of the index structures for increasing dataset sizes, in queries per second.

the start times at all. This is a consequence of Observation 1. In particular, having skewed start times exacerbates the difference in selectivity between the range and duration constraints: when start times are very concentrated on the timeline, the range constraint of a query is satisfied by either most of the intervals or by almost none.

Overall, we observe that indexing first by duration has either better or comparable performance compared to indexing first by time. Therefore, we recommend to choose the former ordering of dimensions when building an RD-INDEX.

6.5 Scalability with Respect to the Input Size

To test the scalability of the index structures, we consider range-duration queries and datasets of increasing size, while maintaining the output size constant. This allows to assess the influence of the input size on the performance without conflating the results with the time required to iterate over larger outputs.

We consider the three real-world datasets, along with a synthetic one with uniform start times and Zipf-distributed durations, which is representative of the distribution of many real-world datasets. Then, we artificially increase their size as follows. Given a dataset and a scale parameter η, let $\bar{t}$ be the span of time covered by all the intervals in the dataset. We make η copies of each interval and shift copy i in time by $i \cdot \bar{t}$, for $i \in [0, \eta)$. The underlying idea is to repeat the temporal patterns of the dataset on a longer time scale, simulating the scenario in which the relation grows over time.

Figure 8 reports for each scale factor the performance of the best configuration of each algorithm. First, we note that in general the relative performance of the data structures does not change at different dataset scales. There are some notable exceptions. The performance of B-TREE degrades by a factor ≈10 from scale 1 to scale 10. The reason is that the B-TREE indexes the durations, and under our synthetic construction the number of intervals associated to each duration increases by the same scale of the dataset. For similar reasons, the performance of RD-INDEX-DT degrades, albeit in a much less pronounced way.

On Webkit, the performance of GRID-FILE and PERIOD-INDEX★ increases with the scale as the dataset: the effect of our synthetic construction in this case is to compensate for the skew in the start times, giving to both data structures the chance of better partitioning the time dimension.

6.6 Insertion Performance

We now focus on the insertion operation. We omit from the comparison PERIOD-INDEX★ (which does not support updates) and HINT (whose implementation on GitHub does not support updates, which are described in the paper [14]), and GRID-FILE, which is a static data structure that requires to know the range of the data beforehand.

For each of the four datasets we considered in the previous sections, we insert intervals into initially empty indices. The expectation is that the insertion performance degrades as the index grows larger. To measure this effect, we insert the intervals in batches of 50 000, measuring the time for each batch in order to be able to estimate the throughput of the insertions as the size of the index grows. We perform two sets of experiments. In the first the intervals are inserted in random order. In the second intervals are inserted by increasing start time, which simulates a natural *append only* scenario for time-related data.

The figures can be found in the extended version of the paper [13]. In most cases, we find that the best performing data structure for the insertion workload is the B-TREE, both when data is presented in random and sorted order. RD-INDEX follows on the second place for most datasets, with the ordering first by time and then by duration usually performing better. For randomly-ordered insertions,

we note that the performance of RD-INDEX tends to slightly decrease as the index size increases. In the more realistic append only scenario, instead, the insertion throughput of RD-INDEX is more stable and tends to remain constant over time. This is expected, since in such a setting only the last column (when the start time is the first dimension being indexed, otherwise the last cell of each column) is ever restructured, requiring very little data to be moved. Furthermore, the performance in this scenario improves compared to the random order of insertions, in particular on the `MimicIII` dataset, and is on par with the B-TREE on all datasets.

7 CONCLUSIONS

RD-INDEX is an index data structure for temporal intervals that allows to answer efficiently range-duration queries. Our approach, which has provable theoretical guarantees, lends itself to a simple and efficient implementation. In particular, its ability to adapt to the distribution of the input data makes it compare favorably with the state of the art on a variety of workloads. In particular, RD-INDEX has superior performance on a vast array of mixed workloads.

A direction of future work is to extend the RD-INDEX to support interval joins [37], thus addressing several needs with a single index. Furthermore, the favorable comparison with the R*-TREE suggests that a promising research direction is to extend the ideas on which RD-INDEX is based to the case of multidimensional spatial data.

ACKNOWLEDGMENTS

This work was supported by a grant from the Autonomous Province of Bozen-Bolzano with research call “Research Südtirol/Alto Adige 2019” (project ISTeP).

REFERENCES

[1] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms* 4, 1 (2008), 9:1–9:30.
[2] Martin Aumüller and Matteo Ceccarello. 2020. Running Experiments with Confidence and Sanity. In *SISAP (LNCS, Vol. 12440)*. Springer, 387–395.
[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. ACM Press, 322–331.
[4] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *SSTD*. ACM, 100–109.
[5] Andreas Behrend, Rainer Manthey, Gereon Schüller, and Monika Wieneke. 2009. Detecting Moving Objects in Noisy Radar Data Using a Relational Database. In *ADBIS (LNCS, Vol. 5739)*. Springer, 286–300.
[6] Andreas Behrend, Philip Schmiegelt, Jingquan Xie, Ronny Fehling, Adel Ghoneimy, Zhen Hua Liu, Eric S. Chan, and Dieter Gawlick. 2014. Temporal State Management for Supporting the Real-Time Analysis of Clinical Data. In *ADBIS (2) (Advances in Intelligent Systems and Computing, Vol. 312)*. Springer, 159–170.
[7] Mark Berg, Marc Kreveld, Mark Overmars, and OtfriedCheong Schwarzkopf. 2000. More Geometric Data Structures. In *Computational Geometry*. Springer Berlin Heidelberg, 211–233.
[8] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2017. Temporal Data Management - An Overview. In *eBISS (Lecture Notes in Business Information Processing, Vol. 324)*. Springer, 51–83.
[9] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. 2006. Multidimensional Aggregation for Temporal Data. In *EDBT (LNCS, Vol. 3896)*. Springer, 257–275.
[10] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *Proc. VLDB Endow.* 10, 11 (2017), 1346–1357.
[11] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *VLDB J.* 30, 4 (2021), 667–691.
[12] Francesco Cafagna and Michael H. Böhlen. 2017. Disjoint interval partitioning. *VLDB J.* 26, 3 (2017), 447–466.
[13] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2022. Indexing Temporal Relations for Range-Duration Queries. *CoRR* abs/2206.07428 (2022). https://doi.org/10.48550/arXiv.2206.07428
[14] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD*. ACM, 1257–1270.
[15] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *SIGMOD*. ACM, 1459–1470.
[16] Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. 2016. Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries. *ACM Trans. Database Syst.* 41, 4 (2016), 26:1–26:46.
[17] Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. 2022. Leveraging range joins for the computation of overlap joins. *VLDB J.* 31, 1 (2022), 75–99.
[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *CoRR* abs/2006.13282 (2020).
[19] Herbert Edelsbrunner. 1980. *Dynamic Rectangle Intersection Searching*. Technical Report 47. Institute for Information Processing, TU Graz, Austria.
[20] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica* 4 (1974), 1–9.
[21] Y. Hayashi and D. L. Paterson. 2011. Strategies for Reduction in Duration of Antibiotic Use in Hospitalized Patients. *Clinical Infectious Diseases* 52, 10 (April 2011), 1232–1240.
[22] Alistair E W Johnson, Tom J Pollard, Lu Shen, Li-Wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific data* 3, 1 (2016), 1–9.
[23] Martin Kaufmann. 2013. Storing and Processing Temporal Data in a Main Memory Column Store. *Proc. VLDB Endow.* 6, 12 (2013), 1444–1449.
[24] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*. ACM, 1173–1184.
[25] Nick Kline and Richard T. Snodgrass. 1995. Computing Temporal Aggregates. In *ICDE*. IEEE Computer Society, 222–231.
[26] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. ACM, 489–504.
[27] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2000. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB*. 407–418.
[28] Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. 2017. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowl. Inf. Syst.* 52, 2 (2017), 341–378.
[29] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43.
[30] D. G. Joakim Larsson and Carl-Fredrik Flach. 2022. Antibiotic resistance in the environment. *Nature Reviews Microbiology* (Nov. 2022), 257–269.
[31] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. 2003. Efficient Algorithms for Large-Scale Temporal Aggregation. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 744–759.
[32] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. ACM, 985–1000.
[33] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (1984), 38–71.
[34] Fabio Persia, Fabio Bettini, and Sven Helmer. 2017. An Interactive Framework for Video Surveillance Event Detection and Modeling. In *CIKM*. ACM, 2515–2518.
[35] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *SSTD (LNCS, Vol. 10411)*. Springer, 125–144.
[36] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *ICDE*. IEEE Computer Society, 1098–1109.
[37] Danila Piatov, Sven Helmer, Anton Dignös, and Fabio Persia. 2021. Cache-efficient sweeping-based interval joins for extended Allen relation predicates. *VLDB J.* 30, 3 (2021), 379–402.
[38] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.
[39] Gereon Schüller, Andreas Behrend, and Rainer Manthey. 2010. AIMS: an SQL-based system for airspace monitoring. In *GIS-IWGS*. ACM, 31–38.
[40] Gereon Schüller, Philip Schmiegelt, and Andreas Behrend. 2012. Supporting Phase Management in Stream Applications. In *ADBIS (LNCS, Vol. 7503)*. Springer, 332–345.
[41] Daniel J. Shapiro, Matthew Hall, Susan C. Lipsett, Adam L. Hersh, Lilliam Ambroggio, Samir S. Shah, Thomas V. Brogan, Jeffrey S. Gerber, Derek J. Williams, Carlos G. Grijalva, Anne J. Blaschke, and Mark I. Neuman. 2021. Short- Versus Prolonged-Duration Antibiotics for Outpatient Pneumonia in Children. *The Journal of Pediatrics* 234 (July 2021), 205–211.e1.
[42] Thatcher Ulrich. 2000. Loose Octrees. In *Game Programming Gems*. Charles River Media, 444–453.