Optimization of Nonsequenced Queries Using Log-Segmented Timestamps

Curtis E. Dyreson
Department of Computer Science, Utah State University, Logan, UT, USA

Abstract

In a period-timestamped, relational temporal database, each tuple is timestamped with a period. The timestamp records when the tuple is "alive" in some temporal dimension. *Nonsequenced semantics* is a query evaluation semantics that involves adding temporal predicates and constructors to a query. We show how to use log-segmented timestamps to improve the efficiency of temporal, nonsequenced queries evaluated using a non-temporal DBMS, i.e., a DBMS that has no special temporal indexes or query evaluation operators. A log-segmented timestamp divides the time-line into segments of known length. Any temporal period can be represented by a small number of such segments. The segments can be appended to a relation as additional columns. The advantage of log-segmented timestamps is that each segment can be indexed using standard database indexes, e.g., a B^+-tree. A query optimizer can use the indexes to generate a lower cost query evaluation plan. This paper shows how to rewrite a query to use the additional columns and evaluates the time cost benefits and space cost disadvantages.

2012 ACM Subject Classification Information systems → Temporal data; Information systems → Relational database query languages

Keywords and phrases Temporal databases, nonsequenced semantics, query evaluation, query performance

Digital Object Identifier 10.4230/LIPIcs.TIME.2023.13

Supplementary Material *Software (Source Code)*: `https://www.usu.edu/cs/people/CurtisDyreson/logsegmented/nonsequenced`

1 Introduction

In a tuple-timestamped, temporal relational database, the lifetime of a tuple in some temporal dimension is recorded using a period timestamp. The period timestamp represents the lifetime using a start time and an end time. Temporal relational database management systems process the timestamps in queries using two commonly recognized semantics for temporal query evaluation: *sequenced* [3] and *nonsequenced* [5]. Sequenced query evaluation, in effect, runs the query in every time instant, while nonsequenced semantics is about the evaluation of explicit temporal predicates, constructors and functions.

We previously showed how to use a different kind of timestamp, which we called a log-segmented timestamp, to implement sequenced semantics for queries in an unmodified relational DBMS [14] and that sequenced semantics can be leveraged to support other kinds of semantics [16]. This paper shows how to use log-segmented timestamps to implement nonsequenced semantics. The primary benefit of doing so is that the log segments can be indexed by non-temporal indexes, and the indexes can be used to (sometimes) lower the cost of query evaluation.

To illustrate nonsequenced query evaluation, consider the query given in Figure 1. The query computes the join on the `dept` attribute between the `Tesco` and `Walmart` relations; the relations are shown in Figure 2. The `OVERLAPS` temporal predicate in the `WHERE` clause determines if the timestamps for the `time` attributes in each relation overlap. The query can be translated by a layer into a Postgres SQL query as shown in Figure 3. Evaluating the query in Figure 3 gives the result in query Figure 2(c).

```
SELECT s.dept, OVERLAPS(r, s)
FROM tesco s, walmart r
WHERE r OVERLAPS s
```

Figure 1 Query to compute the temporal join between two tables.

Data	*Time Metadata*	
Dept	**Start**	**Stop**
Shoe	1	5

(a) Relation `Tesco`

Data	*Time Metadata*	
Dept	**Start**	**Stop**
Shoe	2	3
Shoe	5	6

(b) Relation `Walmart`

Data	*Time Metadata*	
Dept	**Start**	**Stop**
Shoe	2	3
Shoe	5	5

(c) Result of the nonsequenced evaluation of the query in Figure 1.

Figure 2 Example relations.

The focus of this paper is on the cost of query evaluation and whether that cost can be reduced. As an example, consider the evaluation of the query in Figure 3 on relations with 50K tuples. The query evaluation plan generated by the SQL compiler for the query is given in Figure 4. To improve query efficiency in the plan a two attribute index was created (`indexstartstop`) on the time attributes as well as individual indexes on each attribute. The index is used in the nested loops join, but the overall cost of the query (given in the top line of the plan) is 30,587,076.

The key research question addressed by this paper is whether this query evaluation plan can be improved using "off-the-shelf" relational DBMS technology, i.e., not using a specialized temporal index or other modifications of a DBMS. Using the techniques presented in this paper we show how to lower the cost to 1,376,011. The optimizer can choose between the plans to generate the lowest cost query.

This paper makes the following contributions.

- We describe how to extend a relation to store a temporal period using log segments.
- We show how to use log segments to evaluate a nonsequenced temporal predicate.
- We describe experiments with the Postgres DBMS that demonstrate the efficacy of our approach. Our experimental reproducibility package is available.[1]

This paper is organized as follows. The next section gives background material relevant to the paper. Section 3 presents the main technical content of the paper, how to store log segments and use the segments in nonsequenced query evaluation. Evaluation of the technique is given in Section 4 followed by a discussion of related work and a short conclusion with remarks on future work.

[1] `https://www.usu.edu/cs/people/CurtisDyreson/logsegmented/nonsequenced`

```
SELECT s.dept,
       GREATEST(r.time.start, s.time.start) AS start,
       LEAST(r.time.stop, s.time.stop) as stop
FROM tesco s, walmart r
WHERE ((r.start <= s.start AND s.start <= r.stop)
        OR (s.start <= r.start AND r.start <= s.stop))
```

Figure 3 Query to compute the nonsequenced temporal join between two tables.

```
Nested Loop  (cost=228.08..30587076.79 rows=524691358 width=20)
  ->  Seq Scan on empt r  (cost=0.00..1662.00 rows=50000 width=20)
  ->  Bitmap Heap Scan on empt s  (cost=228.08..454.30 rows=10494 width=8)
       Recheck Cond: (((r.start <= start) AND (start <= r.stop))
                      OR ((start <= r.start) AND (r.start <= stop)))
      ->  BitmapOr  (cost=228.08..228.08 rows=11111 width=0)
           ->  Bitmap Index Scan on foostart  (cost=0.00..55.86 rows=5556 width=0)
                 Index Cond: ((start >= r.start) AND (start <= r.stop))
           ->  Bitmap Index Scan on foostartstop  (cost=0.00..166.97 rows=5556 width=0)
                 Index Cond: ((start <= r.start) AND (stop >= r.start))
```

Figure 4 Query execution plan for a temporal join.

2 Preliminaries

In this section we describe background material pertinent to the paper.

2.1 Model of time

This research is orthogonal to assumptions about the time-line, number of temporal dimensions, representations of time, and data model. But for simplicity, we make the following assumptions.

- We use a discrete time-line, with chronons ranging from time $-\infty$ to time ∞.
- There is only one time dimension.
- We assume a relational data model (as either sets or bags of tuples) in which every tuple in every relation is annotated with *temporal metadata* that records the lifetime of the tuple in some time dimension. That is, it is a *tuple-timestamped* model [20].

2.2 Temporal Query Semantics

Sequenced and *nonsequenced* semantics were introduced as different semantics for the evaluation of a temporal operation such as a query or data modification, and both semantics are important [19]. Böhlen and Jensen trace the history and meaning of sequenced semantics [2], but, put simply, sequenced semantics evaluates an operation in each time instant using only the data alive at that time. Nonsequenced semantics, in contrast, means that an operation explicitly references and manipulates the timestamps in the data [5]. In some sense, nonsequenced semantics is the absence of a implicit temporal semantics, only explicit, direct manipulation of the timestamps is supported.

One important benefit of both semantics is that they *reduce* to non-temporal semantics. For sequenced semantics, the reducibility is called *snapshot reducibility* [23] or *S-reducibility* [4]. The temporal semantics is defined in terms of a (presumably easily understood) *slice* of temporal to non-temporal, and the non-temporal semantics of query evaluation (also, well understood).

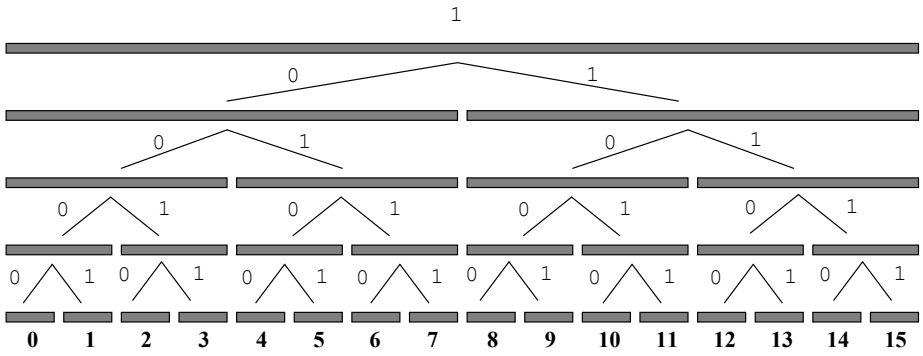

Figure 5 Log segments on a time-line.

Nonsequenced semantics is also reductive. The time information is converted to data, and the non-temporal operation is evaluated on the data. Since time plays no special role in the evaluation, each tuple in the result has no (implicit) time. Instead, the times are manipulated through temporal functions, temporal predicates, and temporal constructors specified in the query. Some of the constructors can convert the data back into time.

Traditionally, the two semantics have been seen as different, though proposals for reconciling the differences exist [16].

2.3 Log-segmented Timestamps

Most temporal database research and implementation uses period timestamps to annotate data with temporal metadata [22]. Period timestamping appends a timestamp to each data item to represent its lifetime. A variation of tuple-timestamped models is *attribute timestamping* where timestamps are appended to each attribute in a tuple rather than to the entire tuple [25].

Period timestamps are a poor fit for architectures that need to partition large data sets into smaller shards to process, e.g., mapreduce architectures [21] or hash joins in a DBMS. Consider, for instance a hash join operation. Data items that have the same join values hash to a common bucket, and the buckets are joined. The strategy is efficient since it ensures that only data items that actually will join are put into a bucket. A temporal join adds a further condition that two data items join only on the times at which they are both alive. For period timestamps this is computed as the temporal intersection of the timestamps. If the intersection is empty, the items do not join since they do not coexist at any point in time. The problem is that periods cannot be directly mapped to buckets in a way that ensures that the items within a bucket temporally intersect. Consider the periods `[1,2]`, `[8,9]`, and `[0,10]`. `[1,2]` and `[8,9]` should be placed in different buckets since they do not intersect, and hence, never represent data that coexists. But `[0,10]` intersects both, it has to be placed into both. Since a period of size n has $n(n+1)/2$ sub-periods that could intersect, every period potentially needs to belong to many buckets.

To address this challenge we developed a *log-segmented timestamp* [13]. The timestamp uses a labelling scheme for pre-determined periods on a time-line. A label is a binary number that has the following meaning.

Table 1 Some example labels for the time-line 0...15.

Label	Period	t_x	t_y
1	0 – 15	0	$15 = 0 + (2^4 - 1)$
10	0 – 7	$0 = 0 * 2^4$	$8 = 0 + (2^3 - 1)$
110	8 – 11	$8 = 1 * 2^3$	$11 = 8 + (2^2 - 1)$
1101	10 – 11	$10 = 1 * 2^3 + 1 * 2^1$	$11 = 10 + (2^1 - 1)$
10011	3 – 3	$3 = 1 * 2^1 + 1 * 2^0$	$3 = 3 + (2^0 - 1)$

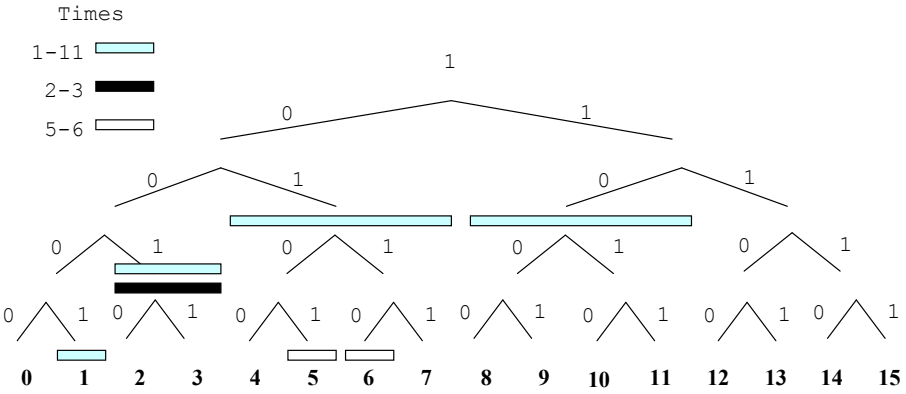

Figure 6 Log segments for the times in the relations in Figure 2 a) and b).

▶ **Definition 1** (Log-segment Label). *Let a (discrete) time-line consist of the times* $t_0, \ldots, t_n$, *where* $n = 2^k - 1$. *Note that* n *can be represented using a binary number of length* k *with each digit set to* `1`. *A label is a binary number,* $b_0 \ldots b_j$, *and* b_0 *is always* `1`. *The label* $1b_1 \ldots b_j$, $j \leq k$, *represents the time period* t_x *to* t_y *where* $t_x = b_1 2^{k-1} + b_2 2^{k-2} + \ldots b_j 2^{k-j}$ *and* $t_y = t_x + (2^{k-j} - 1)$.

The log segments for a time-line from `0` to `15` are depicted in Figure 5. The chronons in the time-line are numbered at the bottom of the figure. Each gray rectangle in the figure is a segment. A label for a segment is the concatenation of `1`'s and `0`'s along the path from the root to a segment. Some example labels are shown in Table 1. Note that only $2n - 1$ of the $(n + 1)(n + 2)/2$ possible periods in the timeline are labelled.

A *log-segmented timestamp* is the minimal set of segments that spans a given period. For example, the log-segmented timestamp representing the period `[3,11]` is `{10011, 101, 110}` (naming the periods `{[3,3], [4,7], [8,11]}`, respectively). The log-segmented timestamps for the times in the relations in Figure 2 a) and b) is graphically depicted in Figure 6.

Log-segmented timestamps have the following properties.

- Comprehensive - A time-line of size n has at most $2n - 1$ labels. Each label will have a maximum length of $1 + \lceil \log_2(n) \rceil$ bits. So a label of 64 bits (the size of a `long long` scalar in C++) can represent a time-line of $2^{63} - 1$ time values, which encompasses a time-line longer than current estimates of the lifetime of the universe to the granularity of microseconds [17].
- Compact - The maximum number of segments in a log-segmented timestamp for a period, $[t_x, t_y]$, is $2 * \lceil \log_2((1 + t_y) - t_x) \rceil$. So assuming 64 bit labels, a log-segmented timestamp has at most 2*64 labels.

Data	*Time Metadata*									
Dept	**Start**	**Stop**	**s1**	**s2**	**s4**	**s8**	**s1x**	**s2x**	**s4x**	**s8x**
...	1	11	10001	1001	101				110	
...	2	3		1001						
...	5	6	10101				10110			

Figure 7 Example segment columns for the periods in Figure 6.

3 Timestamp Representation and Temporal Predicates

To process nonsequenced log-segmented queries, we choose to represent log segments by adding additional columns (attributes) to a table (relation). There are two kinds of additional columns, which we describe in this section: segment columns and prefix columns. It may seem counter intuitive to add columns in order to improve evaluation efficiency, but, in effect, we are trading space for time since the columns that we add will be indexed and the indexes used to lower the time cost.

3.1 Segment Columns

A segment column is a column that stores a log segment as an integer. We observe that a log segmented timestamp has at most two segments with the same length. For instance in Figure 6 there are two segments with the same length, `101` and `110`, in the set of segments for the period `[1-11]`. Hence $2\log_2(n)$ columns are required to store all of the segments. Moreover, each column can be distinguished by the length of the segment that it stores, i.e., a segment with length n can be stored in the $\texttt{segment}_n$ column, and the second segment (if present) in the $\texttt{segment}_n\texttt{x}$ column. An example is shown in Figure 7. It depicts the segments for the log-segmented timestamps in Figure 6. The segment columns (e.g., `s1`) are appended to each row in the table. We assume a timeline for this example of only 16 chronons. An `s16` column is not needed since the entire timeline can be represented by the segments in `s8` and `s8x`.

Any missing segment column value is null, hence the additional columns will be relatively sparsely populated. Most modern DBMSs do not store null values, rather no space is allocated for a null, instead the column is marked as no size in the row header.

3.2 Prefix Columns

The prefix columns record each segment that contains the starting (stopping) chronon of a period. For any given segment, the segment is contained is each segment that is a labelled with a prefix. For instance the segment `1101` is contained within the segments `110`, `11`, and `1`.

Prefix columns are appended to each row in a table to store the prefixes for the start and stop chronons. Each prefix must have a different length, hence the length of the label can be used in the name of the column, e.g., `p4` for a start chronon prefix of length 4 and `p2e` for a stop chronon of length 2. Figure 8 shows the prefixes for the log-segmented timestamps in Figure 6.

Data			*Time Metadata*							
Dept	**Start**	**Stop**	**p1**	**p2**	**p4**	**p8**	**p1e**	**p2e**	**p4e**	**p8e**
...	1	11	10001	1000	100	10	11011	1101	110	11
...	2	3	10010	1001	100	10	10011	1001	100	10
...	5	6	10101	1010	101	10	10110	1011	101	10

Figure 8 Example prefix columns for the periods in Figure 6.

3.3 Reasoning About Chronon Containment

The segment and prefix columns can be used to determine whether a start (or stop) time is contained within a period. Let chronon x have prefixes $p_1, p_2, p_4, \ \ldots, p_n$ and period $[z, y]$ have segments $s_1, s_2, s_4, \ \ldots \ , s_n, s_1x, s_2x, \ldots, s_nx$, then x is contained in $[z, y]$ if

$$\exists i(p_i = s_i \ \vee \ p_i = s_ix)$$

As examples consider the following using the segments of Figure 7 and the prefixes of Figure 8.

- Is 2 contained in `[1-11]`? The segments of `[1-11]` are

Dept	**Start**	**Stop**	**s1**	**s2**	**s4**	**s8**	**s1x**	**s2x**	**s4x**	**s8x**
...	1	11	10001	1001	101				110	

  and the prefixes of 2 are as given below.

Dept	**Start**	**Stop**	**p1**	**p2**	**p4**	**p8**	**p1e**	**p2e**	**p4e**	**p8e**
...	2		10010	1001	100	10				

  Since p2 = s2, it is contained within.
- Is 2 contained in `[5-6]`? The segments of `[5-6]` are

Dept	**Start**	**Stop**	**s1**	**s2**	**s4**	**s8**	**s1x**	**s2x**	**s4x**	**s8x**
...	5	6	10101				10110			

  and the prefixes of 2 are as given below.

Dept	**Start**	**Stop**	**p1**	**p2**	**p4**	**p8**	**p1e**	**p2e**	**p4e**	**p8e**
...	2		10010	1001	100	10				

  There is no i such that pi = si or pi = six, hence it is not contained.

3.4 Temporal Predicates

Log-segmented timestamps give an alternative to using the start and stop times in a period to implement some temporal predicates. Examples include the following.

- `x OVERLAPS y` - If the start chronon in `x` is contained in the period `y` or vice-versa then period `x` overlaps period `y`. Figure 9 shows the SQL to add to the `WHERE` clause to express an `OVERLAPS` predicate, assuming a timeline of 2^{19} chronons. Note that the query could be rewritten using `UNION` or `UNION-ALL` to break up the large disjunctive condition in the `WHERE` clause.

```
WHERE ...
    (x.s1 = y.p1 OR r.s2 = y.p2 OR ... OR x.s19 = y.p19
  OR x.s1 = y.p1e OR r.s2 = y.p2e OR ... OR x.s19 = y.p19e
  OR y.s1 = x.p1 OR y.s2 = x.p2 OR ... OR y.s19 = x.p19
  OR y.s1 = x.p1e OR y.s2 = x.p2e OR .. OR y.s19 = x.p19e)
```

Figure 9 SQL for computing `OVERLAPS` in a time-line of 2^{19}.

```
WHERE ...
  (x.start = y.start AND x.stop <> y.stop AND
   (y.s1 = x.p1e OR y.s2 = x.p2e OR ... OR y.s19 = x.p19e)
  )
```

Figure 10 SQL for computing `STARTS` in a time-line of 2^{19}.

- `x STARTS y` - If `x` and `y` have the same start chronons and different end chronons and the stop chronon of `x` is contained in period `y` then `x` starts `y`. Figure 10 shows the SQL to add to the `WHERE` clause to express a `STARTS` predicate, assuming a timeline of 2^{19} chronons.
- `x CONTAINS y` - If the start and stop chronons of `y` are contained within `x` then `x` contains `y`.

Note that log segments do not provide an alternative for predicates that only involve period endpoints, such as `MEETS`. These predicates can still be evaluated using the start and stop chronons in the timestamp.

3.5 DBMS Implementation

It is highly unlikely that a user could manually manage the prefix and segment columns, for instance, populating these columns when inserting a tuple. To better support users we envision a stratum approach to implementation whereby a user interacts with the DBMS through a layer of middleware layer. The layer provides three key services.

1. Query rewriting - We observe that the schema supports query evaluation on both log-segmented timestamps and normal timestamps (it is not an either-or choice, both can be supported simultaneously). A query will be rewritten by replacing the nonsequenced and sequenced predicates and constructors in the query in two ways. First the query will be rewritten to evaluate with respect to the normal (non-log segmented timestamps). For example an `overlaps` predicate will be replaced with the SQL to compare `start` and `stop` timestamps. Note that this is how non-sequenced query evaluation is usually implemented. Second the query will be rewritten to use the log segmented timestamps. Both rewritten queries will be submitted to the query optimizer to determine which has an (estimated) lower cost, and that query evaluation plan will be chosen.
2. Schema modification - Schema modification statements, e.g., `CREATE TABLE` will be rewritten to manage the prefix and segment columns automatically. All indexes for the additional columns will be created or dropped as needed.
3. Data modification - Data modification statements will be rewritten to manage the prefix and segment columns automatically. Note that computing log segments is a simple calculation that can be done using an SQL function [14].

4 Evaluation

Log segments add columns to a relation and complicate the expression of temporal predicates, but they provide one key benefit: the segment and prefix columns can be indexed and the indexes can be utilized by the query optimizer to lower the time cost of query evaluation. This section describes an experimental evaluation of temporal predicates using log segmented timestamps.

4.1 Experiment Environment

The experiments were run on an Intel Core i7 CPU, 1.8 GHz clock speed, 16 GB of memory and 1 TB SSD drive running Windows 10 Pro 64-bit as the operating system. We used Postgres, version 14, and did not change any installation or configuration parameters from the standard (default) installation.

4.2 Schema for Experiments

We tested with two schemas: `periodStamped` and `segmentStamped`. The `periodStamped` schema has one table, an `Employee` table with the schema given below.

```
Employees(id, name, department, start, stop)
```

The `id` column is the primary key of the table (the snapshot versus temporal key is not relevant to the experiments) and an integer type, the `name` and `department` columns are string types, and the `start` and `stop` columns are integers. The `segmentStamped` schema has one table, an `Employee` table with the schema given below.

```
Employees(id, name, department, start, stop,
          s1, s2, ..., s19, s1x, s2x, ..., s19x,
          p1, p2, ..., p19, p1e, p2e, ..., p19e)
```

The added segment and prefix columns are integer types. We chose to represent a log segment using the time of the first chronon in the segment.

4.3 Database Generation

We synthetically generated a database for each of the schemas. We used 100 different departments and 90% (of the total number of tuples) different names when populating the table. We chose a timeline of 2^{19} (enough to represent a span of 60 years to a granularity of seconds) and used timestamps randomly chosen within the timeline and of random length (from 1 to 2^8). We used test cases of 10000, 20000, 30000, 40000, and 50000 tuples and created one column indexes for every timestamp column (`start` through `p19e`) as well as a two column index on the `start` and `stop` columns. The resulting database sizes are shown in Figure 11. The log segmented tables are roughly twice the size of the period stamped tables, but the indexes for the log segmented tables approximately quadruple the storage cost.

4.4 Measuring Query Cost

To mitigate the impact of database buffering, we used the query cost as estimated by the Postgres query optimizer using `EXPLAIN`. The optimizer computes the cost in units that do not have an exact correspondence to running time, i.e., a cost of 100 does not mean 100 ms of time taken to evaluate a query, but rather are used to determine cheaper versus more expensive queries.

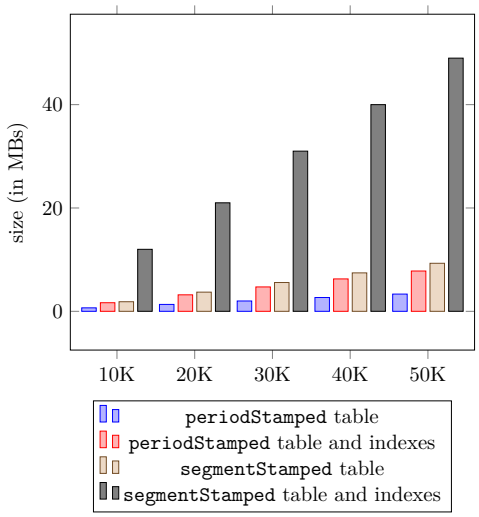

Figure 11 Size of data in database (in MBs).

We also measured actual query time using `EXPLAIN ANALYZE`. We took the minimum cost query over five runs (the variance was inconsequential). The measurements were taken with respect to a warm cache.

4.5 Predicate Evaluation

We measured the cost of three predicates on a fully temporal join, i.e., joining the two relations only on the temporal attributes. The first experiment measure the cost of overlaps. The results are shown in Figure 12. The log-segmented cost is estimated by the compiler to be much lower than that of the period stamped relations. The reason is a different query execution plan. The query execution plan for the period stamped relations was given previously (see Figure 4). Figure 16 shows the relevant part of the log segmented query execution plan. The compiler generates an efficient plan that uses bitmap indexes for matches between segment and prefix columns, yielding a lower cost query plan. The actual timings of the queries shown in Figure 14 show that the query optimizer produced an accurate estimate, and that the bitmap index use does speed up queries.

The second experiment measures the cost of contains. The results are shown in Figure 13. Contains is slightly more efficient for both period and log-segmented timestamped relations. Note that in the log-segmented plot the cost of the 50K join is less than that of the 40K cost. This reflects a change in the optimization strategy chosen by the query optimizer; at 50K tuples the optimizer chooses a parallel scan so gets some performance improvement. Figure 15 shows that the measured query times have the same profile as the estimates produced by the query optimizer.

The third experiment measures the cost of starts. The results are shown in Figure 17. Note that overall the costs are orders of magnitude lower than the other two predicates. This is because both the period and log segmented queries use an equality comparison on the start time, e.g., `r.start = s.start`, together with a test to determine whether the stop time is less. The start time condition can take advantage of the start index for both kinds of timestamp, and this in effect determines the cost of the query. The cost of testing end stop time is slightly worse for the log segmented timestamp, which increases its cost slightly. Note that the cost of the 50K case is better than the 40K case for the log segmented timestamps

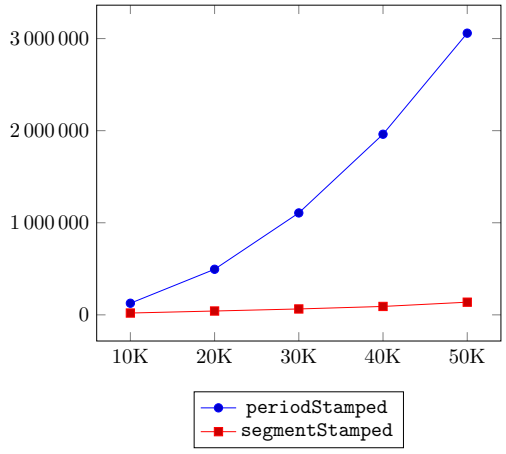

Figure 12 Optimizer estimate for overlaps.

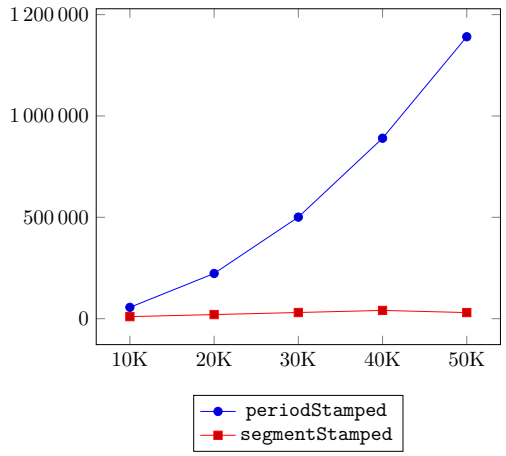

Figure 13 Optimizer estimate for contains.

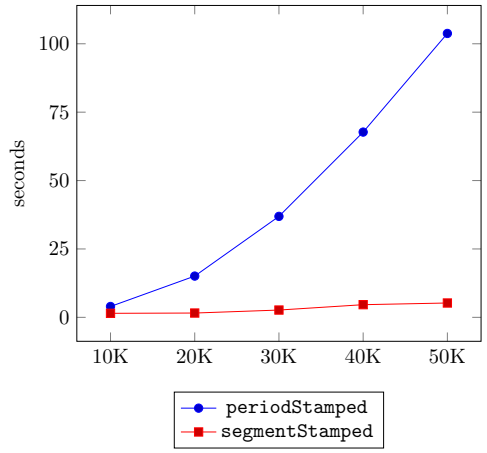

Figure 14 Measured Time for overlaps.

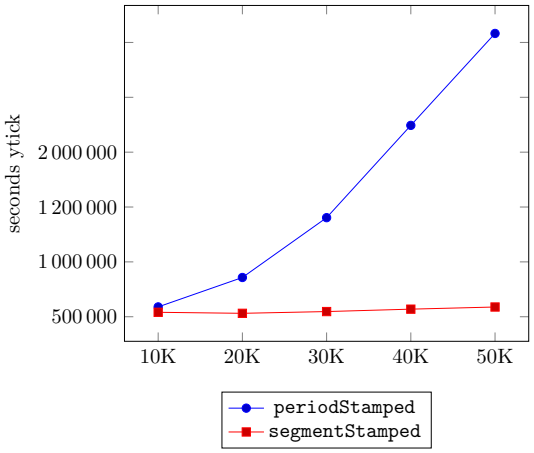

Figure 15 Measured Time for contains.

due to parallelization in the query execution plan. Figure 18 shows that period timestamped query have slightly lower times than the log-segmented timestamped, as predicted by the query optimizer.

4.6 Discussion of Results

Appending columns to a relation to store log segments and prefixes of the start and stop chronons effectively doubles the size of a relation with few data columns. Adding indexes on the segment and prefix columns further increases the cost. But, in practice relations with 10 to 100 data columns are more common, so the storage cost difference would often be less in real-world situations. In some cases the query optimizer can use the added columns and indexes to generate a lower cost query evaluation plan at the cost of increasing the size and complexity of the `WHERE` clause predicate. But in many cases using the start and stop times and indexes offers a better plan as shown by the third experiment (the starts experiment). Utilizing log segments can be seen as a potential optimization technique that increases the space of potential plans, and the query optimizer can examine other constraints in the query to choose the best, lowest cost plan.

```
Nested Loop  (cost=12.13..102716.47 rows=120094 width=20)
  ->  Seq Scan on empt r  (cost=0.00..417.00 rows=10000 width=176)
  ->  Bitmap Heap Scan on empt s  (cost=12.13..18.11 rows=18 width=84)
  Recheck Cond: ((s1 = r.p1) OR (s2 = r.p2) ... OR (s262144 = r.p262144x)
    OR (s524288 = r.p524288x))
    ->  BitmapOr  (cost=12.13..12.13 rows=18 width=0)
      ->  Bitmap Index Scan on foos1  (cost=0.00..0.30 rows=1 width=0)
                                Index Cond: (s1 = r.p1)
      ->  Bitmap Index Scan on foos2  (cost=0.00..0.30 rows=1 width=0)
                                Index Cond: (s2 = r.p2)
      ->  Bitmap Index Scan on foos4  (cost=0.00..0.30 rows=1 width=0)
      ...
      ->  Bitmap Index Scan on foos524288x  (cost=0.00..0.30 rows=1 width=0)
                                Index Cond: (s524288 = s_1.p524288)
```

Figure 16 Query execution plan using indexes on the segment and prefix columns for a temporal join.

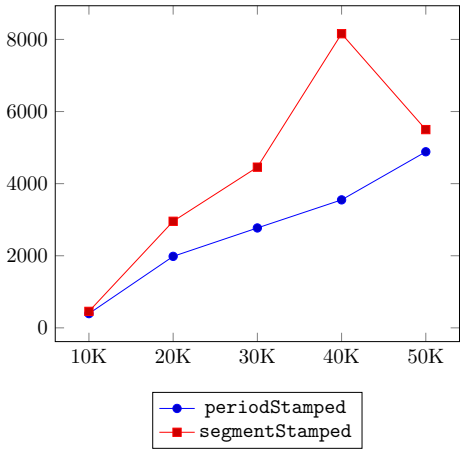

Figure 17 Optimizer for starts.

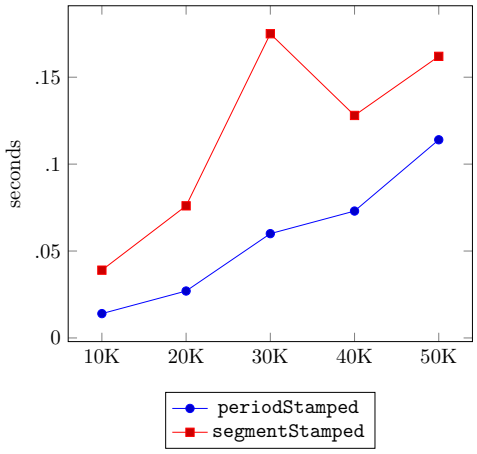

Figure 18 Measured time for starts.

5 Related Work

There are many temporal extensions of query languages, c.f., [7,18,23,24]. This paper focuses on temporal SQL. The extensions have been broadly characterized in various ways but *sequenced* vs. *nonsequenced* distinguishes extensions, in part, by whether the time metadata is manipulated implicitly or explicitly. This paper is about nonsequenced semantics. Temporal languages have also been characterized as *abstract* vs. *concrete* based on whether their syntax and semantics depends on a specific representation of the time metadata [8]. This paper describes an abstract semantics, and proposes a concrete representation to optimize some nonsequenced queries.

Two implementation approaches are common for SQL-like temporal query languages. A *stratum*-approach adds a source-to-source translation layer to translate a query in a temporal extension into an equivalent query in the original, non-extended language [26,27]. Some constructs prove not possible to translate using period timestamps, e.g., sequenced outer join, so the only feasible approach is to extend the DBMS itself [11]. A related approach is to translate to a non-standard variant of SQL [15], in anticipation that SQL will one day evolve to incorporate the variant. We adopt a stratum approach in this paper whereby a nonsequenced

query is translated to a (non-temporal) SQL query and evaluated on a unmodified relational DBMS. We know of no other papers that explore optimization of nonsequenced queries using non-temporal indexes or without making other changes to the DBMS.

Hierarchical partitioning of intervals into smaller segments, similar to log segments, for the purposes of indexing has been explored recently [9]. Our research [13, 14] predates this effort and supports indexing by B-tree indexes.

There are several papers that also support the use of B-tree indexes in evaluating temporal constructors and predicates c.f., [1, 6, 10, 12]. In particular, it was proposed that a range query on a B-tree index combined with a UNION could be used to efficiently compute a non-sequenced join using an overlaps predicate [12]. While we found also found that UNION, or more specifically UNION-ALL, was useful in optimizing queries with an OR predicate in the WHERE clause, but care had to be taken to preserve duplicates or not over-produce duplicates in the query result. The UNION-ALL optimization could also be used for log segmented timestamps which have many OR predicates in overlap joins. One key difference is that we do not use range index queries, rather we use point index queries to evaluate the join. Techniques to augment the DBMS evaluation engine for improved join strategies [1] go beyond the scope of this paper, we focused on not altering the DBMS query evaluation engine.

6 Conclusion and Future Work

The primary contribution of this paper is to show a novel method for optimizing nonsequenced SQL queries. Temporal query languages often extend a non-temporal language by adding temporal predicates, constructors, and functions which directly manipulate the time metadata that annotates the data. A query is said to be nonsequenced if it explicitly includes one of these added temporal features. When a nonsequenced query is evaluated, the nonsequenced part of the query is evaluated against the time metadata, e.g., a temporal overlaps predicate checks if two timestamps overlap in time.

A tuple-timestamped relational database appends to each tuple a period timestamp for each temporal dimension. The start and stop times in the timestamp can be indexed, and often a query execution plan can use the indexes to lower the cost of query execution. This paper proposes adding a log-segmented timestamp to each tuple, in addition to a period timestamp. A log-segmented timestamp divides the time-line into segments of known length. Any temporal period can be represented by a small number of such segments. The segments can be used as an alternative to determining containment of a start or stop chronon within a period. We described how a relation can be extended with segment and prefix columns and how these columns can be used in the nonsequenced evaluation of temporal predicates such as `OVERLAPS`. We experimentally showed that an off-the-shelf relational DBMS can index the segments and the query optimizer can use the indexed segments to generate a lower cost query evaluation plan, though with higher space cost.

In future we plan to continue to investigate log segmented timestamps. We observe that such segments can be used to improve temporal hash joins with a specialized temporal hash join operator that can be added to a DBMS. The idea is that each tuple is first hashed to a data bucket, and if that data bucket becomes full, then further hashed to different time buckets within a data bucket by using log segments as the hash function. A time bucket joins with all time buckets within a data bucket that are prefixes of the segment. A second avenue to explore is prefix-based indexing. In this paper, we proposed precomputing the prefixes and storing them as additional columns, but the prefixes are actually visited in the

traversal of a B^+ tree that stores the segments. By modifying the traversal, it should be possible to avoid precomputation and storage of the prefixes. A third area of future work is temporal constructors and functions. We believe that it is straightforward to articulate temporal constructors, such as the `OVERLAPS` constructor, but have yet to articulate the details. Finally, we are investigating the use of log segments in other temporal query languages such as temporal graph query languages.

References

References

1. Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*, pages 100–109. ACM, 2019. doi:10.1145/3340964.3340965.
2. Michael H. Böhlen and Christian S. Jensen. Sequenced semantics. In *Encyclopedia of Database Systems*, pages 2619–2621. 2009. doi:10.1007/978-0-387-39940-9_1053.
3. Michael H. Böhlen and Christian S. Jensen. Sequenced semantics. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi:10.1007/978-1-4614-8265-9_1053.
4. Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Temporal Statement Modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, 2000. URL: http://portal.acm.org/citation.cfm?id=377674.377665.
5. Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Nonsequenced semantics. In *Encyclopedia of Database Systems*, pages 1913–1915. 2009. doi:10.1007/978-0-387-39940-9_1052.
6. Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. Indexing Temporal Relations for Range-Duration Queries. *CoRR*, abs/2206.07428, 2022. doi:10.48550/arXiv.2206.07428.
7. Cindy Xinmin Chen and Carlo Zaniolo. Sql^{st}: A spatio-temporal data model and query language. In *ER*, pages 96–111, 2000. doi:10.1007/3-540-45393-8_8.
8. Jan Chomicki and David Toman. Abstract versus concrete temporal query languages. In *Encyclopedia of Database Systems*, pages 1–6. 2009. doi:10.1007/978-0-387-39940-9_1559.
9. George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. Hint: A hierarchical index for intervals in main memory, 2021. arXiv:2104.10939.
10. Carlo Combi and Pietro Sala. Interval-based temporal functional dependencies: specification and verification. *Ann. Math. Artif. Intell.*, 71(1-3):85–130, 2014. doi:10.1007/s10472-013-9387-1.
11. Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal Alignment. In *SIGMOD*, pages 433–444, 2012. doi:10.1145/2213836.2213886.
12. Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. Leveraging Range Joins for the Computation of Overlap joins. *VLDB J.*, 31(1):75–99, 2022. doi:10.1007/s00778-021-00692-3.
13. Curtis E. Dyreson. Using CouchDB to Compute Temporal Aggregates. In *18th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 1131–1138. IEEE Computer Society, 2016. doi:10.1109/HPCC-SmartCity-DSS.2016.0159.
14. Curtis E. Dyreson and M. A. Manazir Ahsan. Achieving a Sequenced, Relational Query Language with Log-Segmented Timestamps. In *TIME 2021*, volume 206 of *LIPIcs*, pages 14:1–14:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.TIME.2021.14.

15 Curtis E. Dyreson and Venkata A. Rani. Translating Temporal SQL to Nested SQL. In *23rd International Symposium on Temporal Representation and Reasoning, TIME*, pages 157–166. IEEE Computer Society, 2016. doi:10.1109/TIME.2016.24.

16 Curtis E. Dyreson, Venkata A. Rani, and Amani Shatnawi. Unifying Sequenced and Non-sequenced Semantics. In *22nd International Symposium on Temporal Representation and Reasoning, TIME*, pages 38–46. IEEE Computer Society, 2015. doi:10.1109/TIME.2015.22.

17 Curtis E. Dyreson and Richard T. Snodgrass. Timestamp semantics and representation. *Information Systems*, 18(3):143–166, 1993. doi:10.1016/0306-4379(93)90034-X.

18 Fabio Grandi. T-SPARQL: A TSQL2-like Temporal Query Language for RDF. In *ADBIS*, pages 21–30, 2010. URL: http://ceur-ws.org/Vol-639/021-grandi.pdf.

19 Fabio Grandi, Federica Mandreoli, Riccardo Martoglia, and Wilma Penzo. Unleashing the power of querying streaming data in a temporal database world: A relational algebra approach. *Inf. Syst.*, 103:101872, 2022. doi:10.1016/j.is.2021.101872.

20 C. S. Jensen and C. E. Dyreson (editors). A Consensus Glossary of Temporal Database Concepts - February 1998 Version. In *Temporal Databases: Research and Practice, Lecture Notes in Computer Science 1399*, pages 367–405. Springer-Verlag, 1998.

21 Seyed Nima Khezr and Nima Jafari Navimipour. Mapreduce and its applications, challenges, and architecture: a comprehensive review and directions for future research. *J. Grid Comput.*, 15(3):295–321, 2017. doi:10.1007/s10723-017-9408-0.

22 R. T. Snodgrass. Introduction to TSQL2. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 2, pages 19–31. Kluwer Academic Publishers, 1995.

23 Richard T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987. doi:10.1145/22952.22956.

24 Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

25 A. U. Tansel. Modelling temporal data. *Information and Software Technology*, 32(8):514–520, October 1990.

26 Kristian Torp, Christian S. Jensen, and Michael H. Böhlen. Layered temporal DBMS: concepts and techniques. In *DASFAA*, pages 371–380, 1997.

27 Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. In *IDEAS*, pages 4–13, 1998. doi:10.1109/IDEAS.1998.694346.