Achieving a Sequenced, Relational Query Language with Log-Segmented Timestamps

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

M. A. Manazir Ahsan
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. *Sequenced semantics* is a special semantics for evaluating a query in a temporal database. The semantics stipulates that the query must, in effect, be evaluated simultaneously in each time instant using the tuples alive at that instant. Previous research has proposed changes to the query evaluation engine to support sequenced semantics. In this paper we show how to achieve sequenced semantics without modifying a query evaluation engine. Our technique has two pillars. First we use log-segmented timestamps to record a tuple's lifetime. 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. Second, by taking advantage of the properties of log-segmented timestamps, we translate a sequenced relational algebra query to a non-temporal relational algebra query, using the operations already present in an unmodified, non-temporal query evaluation engine. The primary contribution of this paper is how to implement sequenced semantics using log-segmented timestamped tuples in a generic DBMS, which, to the best of our knowledge, has not been previously shown.

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

Keywords and phrases Temporal databases, sequenced semantics, query evaluation, relational algebra

Digital Object Identifier 10.4230/LIPIcs.TIME.2021.14

Acknowledgements The open access publication of this article was supported by the Alpen-Adria-Universität Klagenfurt, Austria.

1 Introduction

A tuple-timestamped, temporal relational database is a relational database in which each tuple is annotated with a period timestamp, that is, a period of time from some start time to some end time. The timestamp is *metadata* about the tuple; it records when the data was "live" in some temporal dimension.

Temporal relational database management systems (TDMBSs) provide special handling for time metadata in queries. For instance, the *timeslice* operation retrieves the data that is alive at a specified time. TDBMSs typically support a wide range of temporal query operations but the most important is arguably *sequenced semantics* [2]. Informally, sequenced semantics states that the meaning of a sequenced query is that it is equivalent to the (non-temporal) query applied to every snapshot of the data, effectively sequenced semantics is akin to running the query simultaneously in every snapshot in the data's history. We previously showed that sequenced semantics sequenced semantics can be leveraged to support other kinds of semantics [11], e.g., nonsequenced semantics [3].

```
SELECT dept
FROM storesGoldCoast
WHERE dept NOT IN (SELECT dept FROM storesRobina)
```

Figure 1 Query to compute the difference between two tables.

Data	*Metadata*
Dept	**Time**
`Shoe`	`[1,11]`

(a) Relation `storesGoldCoast`

Data	*Metadata*
Dept	**Time**
`Shoe`	`[2,3]`
`Shoe`	`[5,6]`

(b) Relation `storesRobina`

Data	*Metadata*
Dept	**Time**
`Shoe`	`[1,1]`
`Shoe`	`[4,4]`
`Shoe`	`[7,11]`

(c) Result of sequenced evaluation of query in Figure 1.

Figure 2 Example relations.

The history of data can span many instants so it is infeasible to actually run a query on each and every snapshot. To support sequenced semantics a TDBMS must evaluate a sequenced query in some other way. Generally sequenced semantics is implemented by modifying the query evaluation engine c.f., [7]. Previously it was thought not possible to perform sequenced queries on an unaltered relational database management system (RDBMS), e.g., using an unaltered installation of MySQL or Postgres.

To illustrate what makes sequenced query evaluation challenging, consider the SQL query given in Figure 1 which computes the difference between the `dept` attribute in two relations, `storesGoldCoast` and `storesRobina` shown in Figure 2. The query evaluates when there were departments in a `storesGoldCoast` relation and no departments with the same name in the `storesRobina` relation (Robina is a small area within the Gold Coast in Australia). The result of the *sequenced evaluation* of the query is shown in Figure 2(c). What makes the computation complicated is that no single pairing of tuples from the relations computes each tuple in the result, i.e., it cannot be produced by a Cartesian product of the two relations. For instance, we can only figure out the timestamp of the second tuple in the result `[4,4]` by determining that `[2,3]` and `[5,6]` leaves a gap of `[4,4]` within `[1,11]` and that there is no other tuple in `storesRobina` that overlaps `[4,4]`. When moving to the extended relational algebra or SQL, (sequenced) temporal grouping and aggregation, and some subqueries, e.g., `NOT IN` subqueries, are similarly problematic.

In this paper we show how it is possible to translate a sequenced query into a non-temporal query. The translation uses a kind of timestamp that we describe in Section 3. We focus on relational algebra as an example of a complete query language that is widely-known, easy to describe, has a procedural semantics, and provides the basic operations to implement an SQL query evaluation engine. We give a translation of sequenced relational algebra to non-temporal relational algebra in Section 4.

2 Related Work

This paper extends previous research in the area of temporal query languages, There are many temporal extensions of query languages, c.f., [4, 8, 13, 16, 17]. These extensions are designed to add to, rather than change or modify, the prior syntax and semantics of a language. The extensions have been broadly characterized in various ways. *Sequenced* vs. *nonsequenced* distinguishes extensions, in part, by whether the time metadata is manipulated implicitly or explicitly. 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 [5].

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 [19, 20]. 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 [7]. In general, sequenced semantics cannot be directly supported in standard SQL because some of the needed operations are not part of SQL, hence the second strategy extends the DBMS to support additional operations for sequenced semantics. A related approach is to translate to a non-standard variant of SQL [10]. To the best of our knowledge this is the first paper to implement sequenced semantics by translating to standard relational algebra. The translation supports implementation in garden-variety, unaltered relational DBMSs, e.g., MariaDB, Postgres, etc.

Finally, hierarchical partitioning of intervals into smaller segments, similar to log segments, for the purposes of indexing has been explored recently [6]. Our research [9] predates this effort and supports indexing by B-tree indexes.

3 Log-segmented Timestamps

Most temporal database research and implementation uses period timestamps to annotate data with temporal metadata [15]. Period timestamping appends a timestamp to each data item to represent its lifetime. Research has also explored *coalesced* period timestamping in which value-equivalent tuples must have maximally disjoint periods [1]. Another way to represent a coalesced timestamp is with a temporal element, which is a set of disjoint periods [14]. Since a temporal element is a set, it can only be directly stored in a non-1NF data model. 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 [18].

Period timestamps are a poor fit for architectures which need to partition large data sets into smaller shards to process, e.g., mapreduce architectures. Consider, for instance a join operation. Hash-join is usually a good strategy for mapreduce. The mapreduce hash-join maps data items that have the same join values to a common shard, and then joins the items in the shard. The strategy is efficient since it ensures that only data items that actually will join are put into a shard. A *sequenced (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 shards in a way that ensures that the items within a shard temporally intersect. Consider the periods `[1,2]`, `[8,9]`, and `[0,10]`. `[1,2]` and `[8,9]` should be placed in different shards 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^2 sub-periods that could intersect, every period potentially needs to belong to many shards.

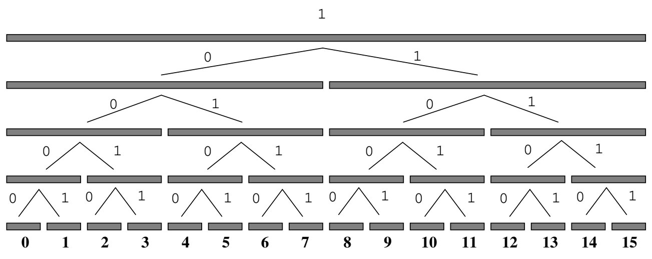

Figure 3 Log segments on a time-line.

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	$3 = 1 * 2^1 + 1 * 2^0$	$3 = 3 + (2^0 - 1)$

To address this challenge we developed a *log-segmented timestamp* [9]. 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.

▶ **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 3. 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^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 5. Figure 4 shows the log-segmented tuples for the relations in Figure 2.

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 [12].
- Compact – The maximum number of segments in a log-segmented timestamp for a period, `[`t_x`,`t_y`]`, is $\lceil \log_2((1 + t_y) - t_x) \rceil$. So assuming 64 bit labels, a log-segmented timestamp has at most 64 labels.

Data	*Metadata*
Dept	**Time**
`Shoe`	`10001`
`Shoe`	`1001`
`Shoe`	`101`
`Shoe`	`110`

(a) Relation `storesGoldCoast`

Data	*Metadata*
Dept	**Time**
`Shoe`	`1001`
`Shoe`	`10101`
`Shoe`	`10110`

(b) Relation `storesRobina`

Figure 4 Example log-segmented relations.

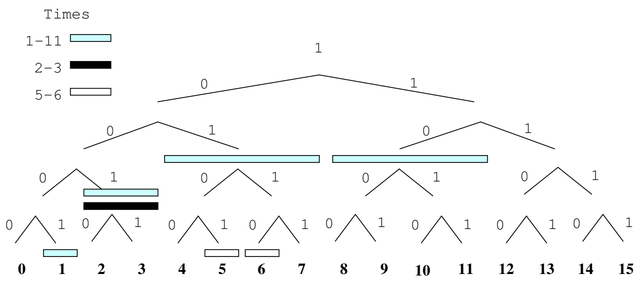

Figure 5 Log segments for the times in the relations in Figure 2 a) and b).

- Efficient for temporal predicates – Predicates in Allen's algebra can be quickly computed. For example for overlaps, given two labels, L_1 and L_2,

$$\texttt{overlaps}(L_1, L_2) = \begin{cases} L_1 & \text{if } L_2 \text{ is a prefix of } L_1 \\ L_2 & \text{if } L_1 \text{ is a prefix of } L_2 \\ \textit{nothing} & \text{otherwise} \end{cases}$$

- Groups – In temporal aggregation a *membership-constant* period is a period of time when some data items, and only, those data items, belong to a group. In a log-segmented timestamp, a label and all prefixes/suffixes of it describe a membership-constant period. So, assuming a timestamp of length 4 the membership-constant period `1001` includes data timestamped with any prefix. Said differently, if we want to compute an aggregate for the period `1001`, we use the data timestamped with `1001`, `100`, `10`, and `1`. So for a time-line of size n there are $\lceil \log_2(n) \rceil$ segments for a membership-constant period.

4 Relational Algebra

In this section we describe a complete set of relational algebra operators for sequenced semantics with log-segmented timestamps. The algebra is defined in terms of non-temporal relational algebraic operators.

4.1 Sequenced Projection

Log-segmented, sequenced projection, $\pi^{\mathcal{T}}$, for some set of attributes A on relation r is defined as follows.

$$\pi^{\mathcal{T}}_{\bar{A}}(r) = \Phi(\pi_{\bar{A},r.T}(r))$$

Φ is the sequenced duplicate elimination operator, which is needed because projection in relational algebra produces a set of tuples, unlike SQL where the underlying model is a bag of tuples. Sequenced duplicate elimination is simple to define for log-segments since for any pair

Data		*Metadata*
Name	**Dept**	**Time**
Joe	Shoe	10
Fred	Shoe	1001
Jennifer	Shoe	101

Figure 6 Example `Employee` relation.

Data		*Metadata*
Dept	**Floor**	**Time**
Shoe	2	10
Shoe	2	111
Photo	1	101

Figure 7 Example `Departments` relation.

of value-equivalent tuples, t and v, if t's timestamp is temporally during v's timestamp, then t can be removed because it is a duplicate. The sequenced duplicate elimination operator is defined below, where ρ is the relation renaming operator (to give a copy of a relation a unique name), $D(t_1, t_2)$ is the timestamp *during* predicate, $r.T$ ($s.T$) is the timestamp for a tuple in relation r (s), $r.V$ ($s.V$) is the list of non-temporal attributes in r (s), and $\bowtie_{r.V=s.V}$ is a value-equivalent equi-join, i.e., the timestamps are ignored in the join, only the non-temporal values are used.

$$\Phi(r) = r - \pi_{r.V,r.T}(\sigma_{D(r.T,s.T)}(r \bowtie_{r.V=s.V} \rho_s(r)))$$

As an example, consider the relation shown in Figure 6 and the query

$$\pi^{\mathcal{T}}_{\texttt{Dept}}(\texttt{Employees}).$$

First we project the `Dept` attribute, as well as the timestamp metadata yielding a relation with three tuples as shown in Figure 9. Next we eliminate sequenced duplicates, yielding the result in Figure 8. The sequenced duplicate elimination removes the second and third tuples because they are during the first tuple's timestamp and are value-equivalent to the first tuple.

4.2 Sequenced Selection (Restriction)

The next operation is log-segmented, sequenced selection, where P is a predicate for deciding if a tuple is in the result relation.

$$\sigma^{\mathcal{T}}_P(r) = \sigma_P(r)$$

Sequenced selection is straightforward it is the same as non-temporal selection; duplicate elimination is not needed since the relation being selected does not contain duplicates, hence the result of a selection cannot have duplicates.

4.3 Sequenced Cartesian Product

Sequenced Cartesian product similarly cannot produce duplicates, but result tuples only exist at the time given by the intersection of two tuples. In the definition, $O(r.T, s.T)$ is the overlaps temporal predicate, $I(r.T, s.T)$ is the temporal intersection constructor, and $r.V$ ($s.V$) is the list of non-temporal attributes in tuple r (s). Note that the projection operator in the definition is a generalized projection since it constructs a timestamp value not present in the operand relations.

$$r \times^{\mathcal{T}} s = \pi_{r.V,s.V,I(r.T,s.T)}(\sigma_{O(r.T,s.T)}(r \times s))$$

As an example if we take the Cartesian product of the relation in Figure 6 with itself, we end up with the relation in Figure 10.

Data	*Metadata*
Dept	**Time**
Shoe	10

Figure 8 After sequenced duplicate are eliminated.

Data	*Metadata*
Dept	**Time**
Shoe	10
Shoe	1001
Shoe	101

Figure 9 The (non-temporal) projection of the `Dept` attribute, need to eliminate sequenced duplicates.

Data		*Metadata*		
Name	**Dept**	**Name**	**Dept**	**Time**
Joe	Shoe	Joe	Shoe	10
Fred	Shoe	Joe	Shoe	1001
Jennifer	Shoe	Joe	Shoe	101
Joe	Shoe	Fred	Shoe	1001
Fred	Shoe	Fred	Shoe	1001
Joe	Shoe	Jennifer	Shoe	101
Jennifer	Shoe	Jennifer	Shoe	101

Figure 10 Example sequenced Cartesian Produce of the `Employee` relation with itself.

4.4 Sequenced Union

Log-segmented, sequenced union adds duplicate elimination to the result of a non-temporal union.

$$r \cup^{\mathcal{T}} s = \Phi(r \cup s)$$

As an example, consider the union of the `Departments` relation shown in Figure 7 with the `Employees` relation in Figure 6 (or rather the projection of each on the `Dept` attribute) as follows.

$$\pi^{\mathcal{T}}_{\texttt{Dept}}(\texttt{Departments}) \cup^{\mathcal{T}} \pi^{\mathcal{T}}_{\texttt{Dept}}(\texttt{Employees})$$

The projection of the `Employees` relation is in Figure 8 and the projection of the `Departments` relation is shown in Figure 11. The result of the union is shown in Figure 12.

4.5 Sequenced Intersection

Sequenced intersection can be expressed using sequenced Cartesian product, selection, and sequenced projection.

Data	*Metadata*
Dept	**Time**
Shoe	10
Shoe	111
Photo	101

Figure 11 Sequenced projection of the `Departments` relation.

Data	*Metadata*
Dept	**Time**
Shoe	10
Shoe	111
Photo	101

Figure 12 Example of a union operation.

$$r \cap^{\mathcal{T}} s = \pi^{\mathcal{T}}_{r.V}(\sigma_{r.V=s.V})(r \times^{\mathcal{T}} s))$$

Intersection can be computed by first taking the sequenced Cartesian product. From this, for all tuples that have value-equivalent pairs in the underlying relation, it takes the sequenced projection of r's attributes. As an example, consider the intersection of the `Employee` relation with itself. First we take the Cartesian product as shown in Figure 10. Next the selection restricts the result to the first, fifth, and seventh tuples since these tuples have the same departments and employee names. Finally the sequenced projection produces the result shown in Figure 6.

4.6 Sequenced Difference

The problem of sequenced, relational difference was described in Section 1. Log-segmented, sequenced relational difference is somewhat complicated. The operation is defined below assuming $C(t_1, t_2)$ is the temporal contains predicate, $O(t_1, t_2)$ is the temporal overlaps predicate, and $E(t_1, t_2)$ is the temporal equals predicate.

$$r -^{\mathcal{T}} s = \Phi(r_c \cup (r_d - (r_d \ltimes_{r.V=s.V \ \wedge \ (C(r_d.T,s.T) \ \vee \ E(r_d.T,s.T))} s)))$$

where

$r_c = r - (r \ltimes_{r.V=s.V \ \wedge \ O(r.T,s.T)} s)$,

$r_d = \pi_{r.V,\mathbb{P}.T_3}((r \bowtie_{r.V=s.V \ \wedge \ C(s.T,r.T)} s) \bowtie_{r.T=\mathbb{P}.T_1 \wedge s.T=\mathbb{P}.T_2} \mathbb{P}))$, and

$\mathbb{P}(T_1, T_2, T_3)$ is the pre-computed log-segmented difference relation.

First, r_c, is the set of tuples that have no value-equivalent match in s or if they have a value-equivalent match do not overlap in time with any tuple in s. Second, r_d is the tuples in r that have a value-equivalent match in s and a lifetime that is during (excluding equals) the lifetime of the tuple in s, which we will call the *during tuples*. The challenge in computing the during tuples is determining potentially *when* they exist since the time is usually not the time of either the tuple in r or in s, which is why relation $\mathbb{P}$ is needed. $\mathbb{P}$ is the *log-segmented difference* relation. It computes the log-segments, attribute T_3, in the difference between a pair of times T_1 and T_2 and is defined as follows, assuming S is the domain of log segments, $C(t_1, t_2)$ is the temporal contains predicate, and $O(t_1, t_2)$ is the temporal overlaps predicate.

$$\mathbb{P}(T_1, T_2, T_3) = \Phi(\{(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in S \ \wedge \ C(t_1, t_3) \ \wedge \ C(t_1, t_2) \ \wedge \ \neg O(t_2, t_3)\})$$

Figure 13 shows some of the tuples in $\mathbb{P}$. For instance, the difference between `10` and `10001` yields the log-segments in the set (in different tuples) {`101`, `1001`, `10000`}. Observe that in Figure 3 these log segments are a set of log segments that together with `10001` span `10`, and are *coalesced*, i.e., no log segment in the set is contained within some log segment, x, such that x is not in the set and x is contained by `10`.

As an example suppose that we take the difference between the `Employees` relation in Figure 6 and the relation in Figure 14. The result is shown in Figure 16. First `Fred` is in the result unchanged from the `Employee` relation since the time in his tuple, `1001`, does not overlap time `11`. That is, `Fred`'s tuple is in r_c. Second, `Jennifer` is not in the result since her tuple's time, `101`, is contained within the time of her tuple in the difference relation, `10`. `Jennifer`'s tuple is not in r_d (or r_c). Finally, consider `Joe`. His tuple has a value-equivalent match that has a lifetime, `10`, which contains his lifetimes in s, `10001` and `101`. `10 - 10001` is {`101,1001,10000`} while `10 - 101` yields {`100`}. So r_d is the relation shown in Figure 15. From this relation we remove any tuple that is value-equivalent and contains or is equal to a time in the difference relation (Figure 14. The first (`101` is equal to `101`), third (`1001` is equal to `1001`), and fourth tuples (`100` is equal to `100`) are removed yielding only the third tuple to be added to the final result.

T_1	T_2	T_3
	. . .	
10	101	100
10	1001	101
10	1001	1000
10	10001	101
10	10001	1001
10	10001	10000
	. . .	

Figure 13 Some tuples in $\mathbb{P}$.

Data		*Metadata*
Name	**Dept**	**Time**
Joe	Shoe	10000
Fred	Shoe	11
Jennifer	Shoe	10

Figure 14 `Employee` difference relation.

Data		*Metadata*
Name	**Dept**	**Time**
Joe	Shoe	101
Joe	Shoe	1001
Joe	Shoe	10000
Joe	Shoe	100

Figure 15 The during tuples in computing the difference.

Data		*Metadata*
Name	**Dept**	**Time**
Joe	Shoe	10001
Fred	Shoe	1001

Figure 16 Result of the sequenced difference of Figure 6 and Figure 14.

4.7 Sequenced Grouping and Aggregation

Sequenced grouping and aggregation is also possible with log segments, though the process is somewhat complicated. We first give an informal example of sequenced aggregation and group by, and then a formal definition.

Assume that we want to count the number of `Employees` per `Department` over time, i.e., a sequenced aggregation and grouping. Furthermore, assume that our relation has four tuples for the `Clothing` department timestamped with log-segments 1010, 1010, 101, and 1 as shown in Figure 17.

Step 1: Determine log segment fragments Long-lived tuples potentially span many temporal groups. For instance, in the relation in Figure 17, `Freya`'s tuple contains the lifetime of all the other tuples in the relation so should belong to each group, but also to groups not in the lifetimes of those tuples, e.g., `Freya` is present at time `11` while none of the other tuples are (they are all within `10`). So the goal of this step is to split the timestamps to determine coverage with respect to the other timestamps in the relation. We use temporal difference to split the lifetimes, that is for any lifetime that is contained in another, we take the difference. For instance, in our running example, `(Susan, Clothing, 1010)`

Data		*Metadata*
Name	**Dept**	**Time**
Susan	Clothing	1010
Pedro	Clothing	1010
Malik	Clothing	101
Freya	Clothing	1

Figure 17 Example relation for grouping.

Data		*Metadata*
Name	**Dept**	**Time**
Malik	Clothing	1011
Freya	Clothing	1011
Freya	Clothing	11
Freya	Clothing	100

Figure 18 Fragments of lifetimes.

Data		*Metadata*
Name	**Dept**	**Time**
Freya	Clothing	1010
Malik	Clothing	1010
Freya	Clothing	101

Figure 19 Long-lived tuple are potential group members.

	Data		*Metadata*
Count	**Name**	**Dept**	**Time**
4	Susan	Clothing	1010
4	Pedro	Clothing	1010
4	Freya	Clothing	1010
4	Malik	Clothing	1010
2	Malik	Clothing	1011
2	Freya	Clothing	1011
1	Freya	Clothing	100
2	Malik	Clothing	101
2	Freya	Clothing	101
1	Freya	Clothing	11
1	Freya	Clothing	1

Figure 20 Union of the original relation, Figure 18 and Figure 19 with the aggregate computed.

lifetime is contained in that of `(Malik, Clothing, 101)` so we take the difference of `101` and `1010` to get `1011` and so generate the tuple `(Malik, Clothing, 1011)`. We also do the other pairs, `1 - 101` yielding `(Freya, Clothing, 11)` and `(Freya, Clothing, 100)`, and the pair `1 - 1010` yielding `(Freya, Clothing, 1011)` and `(Freya, Clothing, 11)`. The result relation is shown in Figure 18.

Step 2: Add long-lived tuples to contained lifetime groups This step add long-lived tuples to the groups that have lifetimes that are contained within the lifetimes of the long-lived tuple. For instance, in the relation in Figure 17, `Freya`'s tuple contains the lifetime of all the other tuples in the relation so should belong to each group, e.g., `Freya` is present at time `101` and `1010`. The resulting relation is shown in Figure 19.

Step 3: Gather potential group members Form the union of the results of the original relation, Step 1, and Step 2. The result relation is shown in Figure 20 (the relation depicted has the computed aggregates as well, but those will be added in the next step).

Step 4: Group and aggregate Group and aggregate the result of Step 3, pre-pending the aggregate value (computed for the group) to each tuple. The result relation is shown in Figure 20.

Step 5: Remove containing lifetimes Since lifetimes were fragmented in Step 1 to represent smaller periods, this step removes duplicate counts. A duplicate count is for any tuple that has a lifetime that contains that of another tuple in the relation produced in Step 4. For instance, `(2, Mailik, Clothing, 101)` is a duplicate tuple since its lifetime contains the lifetime of another tuple `(4, Freya, Clothing, 1010)`. Hence it has already been counted and should be removed. The result of this step is shown in Figure 21, which is the sequenced count of `Employees` grouped by `Dept`.

The aggregation operator ${}_{\bar{G}}\mathcal{F}^{\mathcal{T}}_{\bar{A}}$, where $\bar{G}$ is a list of grouping attributes and $\bar{A}$ is a list of aggregate functions, is defined below.

${}_{\bar{G}}\mathcal{F}^{\mathcal{T}}_{\bar{A}}(r) = r_5$
where (note: relation r_i is produced by Step i)
$r_1 = \pi_{r.V,\mathbb{P}.T_3}((r \bowtie_{C(r.T,s.T)\ \wedge\ r.\bar{G}=s.\bar{G}} \rho_s(r)) \bowtie_{r.T=\mathbb{P}.T_1 \wedge s.T=\mathbb{P}.T_2} \mathbb{P})$,
$r_2 = \pi_{r.V,s.T}(r \bowtie_{C(r.T,s.T)\ \wedge\ r.\bar{G}=s.\bar{G}} \rho_s(r))$,
$r_3 = r \cup r_1 \cup r_2$, and
$r_4 =_{\bar{r}.G} \mathcal{F}_{\bar{A}}(r_3)$.
$r_5 = r_4 - (r_4 \ltimes_{r.\bar{G}=s.\bar{G}\ \wedge\ C(r_4.T,s.T)} \rho_s(r_4))$

	Data		*Metadata*
Count	**Name**	**Dept**	**Time**
4	Susan	Clothing	1010
4	Pedro	Clothing	1010
4	Freya	Clothing	1010
4	Malik	Clothing	1010
2	Malik	Clothing	1011
2	Freya	Clothing	1011
1	Freya	Clothing	100
1	Freya	Clothing	11

Figure 21 Sequenced count of `Employees` grouped by `Dept`.

4.8 Cost Analysis

The primary disadvantage of log-segmented relational algebra is cost since the log-segmented increases the size of the relations. Note however, that the size cost could be reduced by normalizing a log-segmented relation, that is, by splitting the data and metadata columns into separate tables, with a foreign key from the metadata table into the data table. In this analysis we do not assume such normalization.

Let relation r (s) be a period timestamped relation with N (M) tuples. Representing the relations using log segments increase the size of the relation by a factor of $f = \log_2(k)$ where k is the maximum time (assuming a time domain from 0 to k). Then the relational algebra operators have the following cost.

- Sequenced projection of r: The cost is $\mathcal{O}(fN)$ to project r and $\mathcal{O}((fN)^3)$ to perform duplicate elimination, so the cost is dominated by duplicate elimination.
- Sequenced selection of r: The cost is $\mathcal{O}(fN)$ to scan through the relation.
- Sequenced Cartesian product of r with s: The cost is $\mathcal{O}(f^2NM)$.
- Sequenced Union of r with s: The cost is $\mathcal{O}(fN) + \mathcal{O}(fM) + \wr((f(N+M))^3)$, so the cost is dominated by duplicate elimination.
- Sequenced Intersection of r with s: The cost is $\mathcal{O}((fN) * (fM))$ since the projection and selection can performed as the Cartesian product is computed.
- Sequenced Difference of r minus s: To compute the *during tuples* costs $\mathcal{O}(f^3NM)$ assuming that $\mathbb{P}$ can by dynamically computed, e.g., such as using a table function in Postgres. To compute r_c costs $\mathcal{O}(f^3N^2M)$. The union of r_c with the during tuples and performing the duplicate elimination costs $\mathcal{O}(f^9N^3M^2)$, so the duplicate elimination again dominates the cost.
- Sequenced Grouping and Aggregation: There are five steps. To compute r_1 costs $\mathcal{O}(f^3NM)$. Computing r_2 squares the cost of r_1 and, assuming linear-time union can be performed, the cost of r_3 is $\mathcal{O}((fN)^2)$, which is the maximum possible size of r_2 or r_3. We will assume computing the aggregate can be done in linear time, so the cost of r_5 is $\mathcal{O}((fN)^4)$

Note that the most frequent query operations are projection, selection, and Cartesian product. The cost of selection and Cartesian product are the same as their non-temporal counterparts (except for the increased size of the relation). But unlike temporal periods, log segments can be indexed using a non-temporal index, e.g., a B$^+$-tree, so there are likely significant query optimization opportunities for sequenced queries using standard SQL query optimization techniques involving indexes. Only projection is significantly more expensive, but the cost is largely due to duplicate elimination, which can be thought of as optional in an SQL-based DBMS, which allows duplicates in the data model. The cost of the other operations (except intersection which is the same as the non-temporal cost) is much higher than their non-temporal counterpart (which do not support sequenced semantics, with the additional functionality comes increased cost). But, overall sequenced queries can be supported in a vanilla SQL-based DBMS and we suspect that query optimization combined with standard indexes can achieve reasonable run-time efficiency.

5 Conclusion and Future Work

The primary contribution of this paper is to show how sequenced semantics can be implemented for a relational query language using the non-temporal form of the language. This demonstration means that it is possible to implement sequenced semantics when evaluating queries in a relational DBMS such as MariaDB without having to make any changes to the DBMS.

In this paper we presented sequenced relational algebra by defining its operations entirely in terms of standard relational algebra, lacking any temporal semantics or constructs. The key to the translation is to interpret timestamps in a different way. Rather than taking the standard approach of using period timestamps we chose to timestamp using *log segments*. The log segments are an *a priori* dividing of the time-line into segments such that the segments cover the time-line and form a hierarchy in which smaller segments group into larger segments. The labels on the segments can be used to efficiently and easily determine temporal relationships such as overlaps or contains. We showed how the segments are used in various operations such as sequenced aggregation and grouping.

Future work is focused on implementation. We are currently implementing a sequenced SQL to SQL translator using Postgres. An open question is the impact of the translation on query optimization. That is, can the query optimizer take advantage of indexes for the log segments in the translated queries? We are also investigating the benefits and costs of normalized representation (factoring the metadata into separate tables). We have not yet begin to look at other issues such as implementation of sequenced constraints using log segments, recursive queries, or application to other query languages such as sequenced GraphQL.

References

1 Michael H. Böhlen. Temporal coalescing. In *Encyclopedia of Database Systems*, pages 2932–2936. Springer, 2009. doi:10.1007/978-0-387-39940-9_388.

2 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.

3 Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Nonsequenced 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_1052.

4 Cindy Xinmin Chen and Carlo Zaniolo. Sqlst: A spatio-temporal data model and query language. In *ER*, pages 96–111, 2000. doi:10.1007/3-540-45393-8_8.

5 Jan Chomicki and David Toman. Abstract versus concrete temporal query languages. In *Encyclopedia of Database Systems*, pages 1–6. Springer, 2009. doi:10.1007/978-0-387-39940-9_1559.

6 George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. Hint: A hierarchical index for intervals in main memory, 2021. arXiv:2104.10939.

7 Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal Alignment. In *SIGMOD*, pages 433–444, 2012. doi:10.1145/2213836.2213886.

8 Curtis E. Dyreson. Observing Transaction-Time Semantics with TTXPath. In *WISE (1)*, pages 193–202, 2001. doi:10.1109/WISE.2001.996480.

9 Curtis E. Dyreson. Using couchdb to compute temporal aggregates. In *18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016, Sydney, Australia, December 12-14, 2016*, pages 1131–1138. IEEE Computer Society, 2016. doi:10.1109/HPCC-SmartCity-DSS.2016.0159.

10 Curtis E. Dyreson and Venkata A. Rani. Translating temporal SQL to nested SQL. In Curtis E. Dyreson, Michael R. Hansen, and Luke Hunsberger, editors, *23rd International Symposium on Temporal Representation and Reasoning, TIME 2016, Kongens Lyngby, Denmark, October 17-19, 2016*, pages 157–166. IEEE Computer Society, 2016. doi:10.1109/TIME.2016.24.

11 Curtis E. Dyreson, Venkata A. Rani, and Amani Shatnawi. Unifying sequenced and non-sequenced semantics. In Fabio Grandi, Martin Lange, and Alessio Lomuscio, editors, *22nd International Symposium on Temporal Representation and Reasoning, TIME 2015, Kassel, Germany, September 23-25, 2015*, pages 38–46. IEEE Computer Society, 2015. doi:10.1109/TIME.2015.22.

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

13 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.

14 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.

15 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.

16 Richard T. Snodgrass. The Temporal Query Language TQuel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987. doi:10.1145/22952.22956.

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

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

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

20 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.