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

M. A. Manazir Ahsan

Utah State University

ACHIEVING A SEQUENCED, RELATIONAL QUERY LANGUAGE WITH LOG-SEGMENTED TIMESTAMPS

by

M A Manazir Ahsan

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Dan Watson, Ph.D. Committee Member
Nick Flann, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Interim Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2021

ABSTRACT

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

by

M A Manazir Ahsan, Master of Science

Utah State University, 2021

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

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.

(56 pages)

PUBLIC ABSTRACT

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

M A Manazir Ahsan

In a relational temporal database, typically each row of each table has a period timestamp to indicate the lifetime of that row. In order to evaluate a query in a temporal database, sequenced semantics comes into play. The semantics stipulates that the query must be evaluated simultaneously in each time instant using the data rows available at that point of time. Existing researches have proposed changes in the query evaluation engine to achieve sequenced semantics. In this paper we show a way to support sequenced semantics without modifying the query engine. We propose a noble construction *log-segmented label* to represent the lifetime and replace the period timestamp from each row with a log-segmented label that signifies when the tuple is alive. Then we translate a sequenced query to a non-temporal query by utilizing the properties of log-segmented label. The translated query has only operations already available in a typical relational database making the query readily executable in an unaltered installation of the database. Thus the sequenced query inevitably runs and retrieve data without changing query evaluation engine. Finally our implementation using Java language, ANTLR parser generator and PostgreSQL database demonstrates the feasibility of the proposed mechanism, which, to the best of our knowledge, has not been previously shown.

We will come back to it soon...

ACKNOWLEDGMENTS

My supervisor helped me at each stage of this research. I gratefully acknowledge it. I am also grateful to all those people whose helping hands made my life easier so that I could concentrate on my research. Above all, total planner is the key reason behind all the blessings I have.

M A Manazir Ahsan

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . iv

ACKNOWLEDGMENTS . . . vi

LIST OF TABLES . . . ix

LIST OF FIGURES . . . x

1 INTRODUCTION . . . 1

2 RELATED WORK . . . 4

3 LOG-SEGMENTED TIMESTAMPS . . . 7

4 RELATIONAL ALGEBRA . . . 11
4.1 SEQUENCED PROJECTION . . . 11
4.2 SEQUENCED SELECTION . . . 12
4.3 SEQUENCED CARTESIAN PRODUCT . . . 12
4.4 SEQUENCED UNION . . . 13
4.5 SEQUENCED INTERSECTION . . . 14
4.6 SEQUENCED DIFFERENCE . . . 14
4.7 SEQUENCED GROUPING AND AGGREGATION . . . 16
4.8 COST ANALYSIS . . . 19

5 IMPLEMENTATION . . . 22
5.1 QUERY TRANSFORMATION . . . 23
5.1.1 SEQUENCED PROJECTION . . . 24
5.1.2 SEQUENCED SELECTION . . . 26
5.1.3 SEQUENCED CARTESIAN PRODUCT . . . 26
5.1.4 SEQUENCED UNION . . . 27
5.1.5 SEQUENCED INTERSECTION . . . 27
5.1.6 SEQUENCED DIFFERENCE . . . 28
5.1.7 SEQUENCED AGGREGATION AND GROUPING . . . 30
5.2 TESTING . . . 34

6 CONCLUSION AND FUTURE WORK . . . 35

REFERENCES . . . 36

APPENDICES . . . 38
A SQL Listing for Testing . . . 39

CURRICULUM VITAE 45

LIST OF TABLES

Table Page

3.1 Some example labels for the time-line 0. . .15 8

LIST OF FIGURES

Figure Page

1.1 Query to compute the difference between two tables 2

1.2 Example relations 2

3.1 Log segments on a time-line 8

3.2 Example log-segmented relations 9

3.3 Log segments for the times in the relations in Figure 1.2 a) and b) 9

4.1 Example `Employee` relation 12

4.2 Example `Departments` relation 12

4.3 After sequenced duplicate are eliminated 13

4.4 The (non-temporal) projection of the `Dept` attribute, need to eliminate sequenced duplicates 13

4.5 Example sequenced Cartesian Produce of the `Employee` relation with itself . 13

4.6 Sequenced projection of the `Departments` relation 14

4.7 Example of a union operation 14

4.8 Some tuples in $\mathbb{P}$ 16

4.9 `Employee` difference relation 16

4.10 The during tuples in computing the difference 16

4.11 Result of the sequenced difference of Figure 4.1 and Figure 4.9 16

4.12 Example relation for grouping 17

4.13 Fragments of lifetimes 17

4.14 Long-lived tuple are potential group members 17

4.15 Union of the original relation, Figure 4.13 and Figure 4.14 with the aggregate computed 17

4.16 Sequenced count of `Employees` grouped by `Dept` 19

5.1 System Design. 22

5.2 Transformation of Sequenced Projection 25

5.3 Transformation of Sequenced Selection 25

5.4 Transformation of Sequenced Cartesian Product 26

5.5 Transformation of Sequenced Union 27

5.6 Transformation of Sequenced Intersection 28

5.7 Temporal Difference between two labels. 29

5.8 Transformation of Sequenced Set Difference 30

CHAPTER 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* [1]. 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 can be leveraged to support other kinds of semantics [1, 2], nonsequenced semantics [3].

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., [4]. 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 1.1 which computes the difference between the `dept` attribute in two relations, `storesGoldCoast` and `storesRobina` shown in Figure 1.2. 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

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

Fig. 1.1: 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 1.1.

Fig. 1.2: Example relations

Australia). The result of the *sequenced evaluation* of the query is shown in Figure 1.2(c). What makes the computation complicated is that no single pairing of tuples from the relations computes each tuple in the result, 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, `NOT IN` subqueries, are similarly problematic.

In this research 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 chapter 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 chapter 4. We describe how to translate sequenced SQL queries into non-sequenced SQL and report on some experiments that measure the cost in the same. We present our implementation details in the chapter 5.

CHAPTER 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., [5–10]. 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 [11].

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 [12,13]. 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 [4]. 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 [14]. 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.

Researchers accumulated a wide range of concepts of temporal databases interest along with their definitions, explanations and discussion of the given names [15]. They identified three primary kinds of time: transaction, valid, and user-defined time. Transaction time defines the time in which an event is alive in a database. More precisely, the database time in between insertion and deletion of an entity is the transaction time. Valid time is the time of existence of an event in the real world. Both the transaction and the valid time are the example of metadata that means, they are data about a data stored in a database. On the contrary, user-defined time is a piece of data of time type. This is a time value of an event, in other words, it is a data that happens to be time. An example could draw a fine line between their differences.

Consider, the fact that Jack Sparrow was born in 1988, had started undergrad education at Utah State University from 2006 to 2009. The university inserted his information into the database in 2008 after adopting a new information system and continued to keep his data for life long. One way to model the information system that contains Mr. Sparrow's study records is to define the transaction time which is the span of his undergrad study [2008-until changed], since the data has been inserted on 2008 and not yet been deleted [16]. On the other hand, the valid time of study which is real world time is [2006-2009]. Besides, birthday will be a piece of information about Mr. Sparrow which happens to be a time and we call it user defined time. User defined time is an attribute of an entity but the transaction and/or valid time are the metadata that changes the semantics of data and the way we executes any operation over the data. Despite their difference, any event could happen monetarily, i.e., all the three times could be represented with the same starting and ending timestamps (like, time of a financial transaction).

Even though the information stored in a database continues to grow with the addition of new data, these growth is are considered as modifications to the state, with outdated data being updated or deleted from the database. Thus the content of the database illustrates the current state or the snapshot of the business being modelled. In contrast, temporal represents the progression of states of a business over a period. Hence, in temporal database, changes are considered as additions to the information in the database without deletion or modification of existing content. Meaning that temporal database preserves multiple snapshot of a business over time. Due to having multiple snapshots the extraction of data from a temporal database requires a query language with special syntax and semantics [8].

Several languages of that kind are available in the literature, TQuel, Statement Modifiers are a few of them. Some researchers designs and implements the temporal query for OLAP (On Line Analytics Processing) [17, 18].

Temporal database researchers codifies a set of requirements (desiderata) that directly defines the syntax and semantics of temporal extension of any non-temporal query language. One research group introduces the concept of Statement Modifiers that shows a way of systematically adding temporal feature to an arbitrary non-temporal query language [19]. Statement modifiers are applicable to all data retrieval statement, modification statement, integrity checking statements, or other data and metadata manipulation statements.

It proposed saptio-temporal data model and query language with a view to minimizing the required extensions in a relational language. The cornerstone of the article are the directed triangulation representation and point based representation for spatial data temporal data respectively. In order to achieve efficiency in this model, it defined conceptual and physical representations and a mapping between them. User defined function/aggregator, in addition to syntactic and semantic notions available in the modern query language, is necessary for the implementation of the spatio-temporal query. Nonetheless, user defined function makes the implementation relatively slower and the model needs to extend the existing SQL engine [5].

CHAPTER 3
LOG-SEGMENTED TIMESTAMPS

Most temporal database research and implementation uses period timestamps to annotate data with temporal metadata [10]. 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 [20]. Another way to represent a coalesced timestamp is with a temporal element, which is a set of disjoint periods [21]. 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 [22].

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 $\frac{n(n+1)}{2}$ sub-periods that could intersect, every period potentially needs to belong to many shards.

To address this challenge we developed a *log-segmented timestamp* [23]. The timestamp

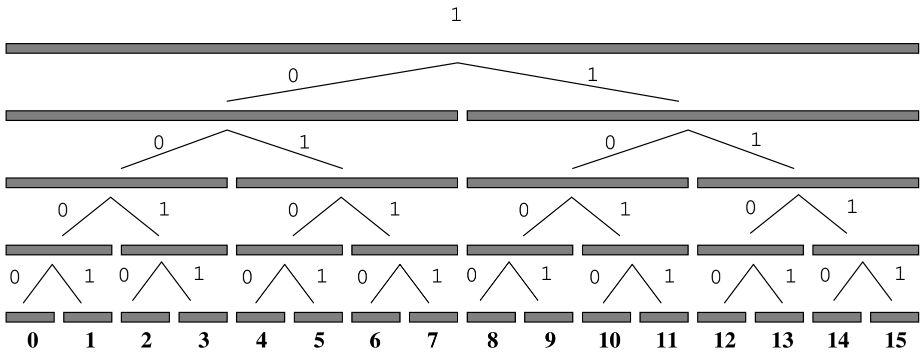

Fig. 3.1: Log segments on a time-line

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

Table 3.1: Some example labels for the time-line 0. . .15

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-segmented 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 3.1. 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 3.1. 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,`

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

(a) Relation `storesGoldCoast`

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

(b) Relation `storesRobina`

Fig. 3.2: Example log-segmented relations

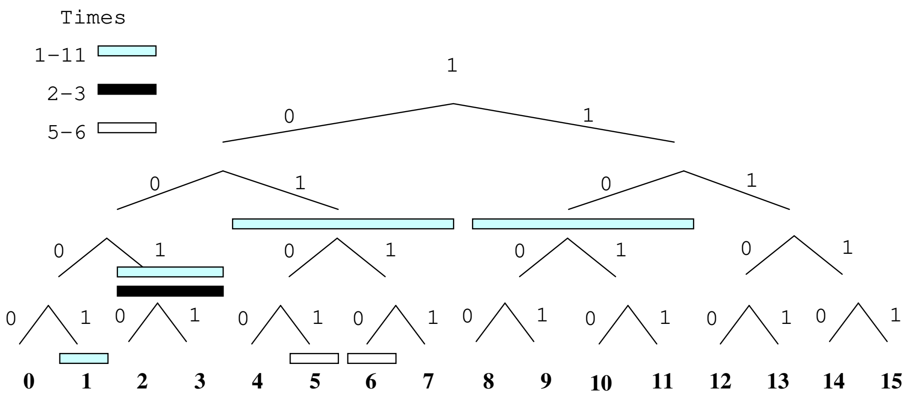

Fig. 3.3: Log segments for the times in the relations in Figure 1.2 a) and b)

`110`} (naming the periods {`[3,3]`, `[4,7]`, `[8,11]`}, respectively). The log-segmented timestamps for the times in the relations in Figure 1.2 a) and b) is graphically depicted in Figure 3.3. Figure 3.2 shows the log-segmented tuples for the relations in Figure 1.2.

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 [24].

- 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.

- 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.

CHAPTER 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 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, 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 4.1 and the query

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

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

Fig. 4.1: Example `Employee` relation

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

Fig. 4.2: Example `Departments` relation

First we project the `Dept` attribute, as well as the timestamp metadata yielding a relation with three tuples as shown in Figure 4.4. Next we eliminate sequenced duplicates, yielding the result in Figure 4.3. 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

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

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

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.

Data Dept	*Metadata* Time
Shoe	10

Fig. 4.3: After sequenced duplicate are eliminated

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

Fig. 4.4: The (non-temporal) projection of the `Dept` attribute, need to eliminate sequenced duplicates

Data Name	Dept	*Metadata* 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

Fig. 4.5: Example sequenced Cartesian Produce of the `Employee` relation with itself

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

As an example if we take the Cartesian product of the relation in Figure 4.1 with itself, we end up with the relation in Figure 4.5.

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 4.2 with the `Employees` relation in Figure 4.1 (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 4.3 and the projection of the `Departments` relation is shown in Figure 4.6. The result of the union is shown in Figure 4.7.

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

Fig. 4.6: Sequenced projection of the `Departments` relation

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

Fig. 4.7: Example of a union operation

4.5 SEQUENCED INTERSECTION

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

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

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 4.5. 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 4.1.

4.6 SEQUENCED DIFFERENCE

The problem of sequenced, relational difference was described in chapter 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 4.8 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 3.1 these log segments are a set of log segments that together with `10001` span `10`, and are *coalesced*, 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 4.1 and the relation in Figure 4.9. The result is shown in Figure 4.11. 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 4.10. From this relation we remove any tuple that is value-equivalent and contains or is equal to a time in the difference relation (Figure 4.9. The first (`101` is equal

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

Fig. 4.8: Some tuples in $\mathbb{P}$

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

Fig. 4.9: `Employee` difference relation

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

Fig. 4.10: The during tuples in computing the difference

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

Fig. 4.11: Result of the sequenced difference of Figure 4.1 and Figure 4.9

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.

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 4.12.

Step 1: Determine log segment fragments Long-lived tuples potentially span many temporal groups. For instance, in the relation in Figure 4.12, `Freya`'s tuple contains the lifetime of all the other tuples in the relation so should belong to each group,

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

Fig. 4.12: Example relation for grouping

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

Fig. 4.13: Fragments of lifetimes

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

Fig. 4.14: 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

Fig. 4.15: Union of the original relation, Figure 4.13 and Figure 4.14 with the aggregate computed

but also to groups not in the lifetimes of those tuples, `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)` 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 4.13.

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 4.12, `Freya`'s tuple contains the lifetime of all the other tuples in the relation so should belong to each group, `Freya` is present at time `101` and `1010`. The resulting relation is shown in Figure 4.14.

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 4.15 (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 4.15.

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

	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`

Fig. 4.16: Sequenced count of `Employees` grouped by `Dept`

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 4.16, which is the sequenced count of `Employees` grouped by `Dept`.

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

$${}_{\bar{G}}F^{\mathcal{T}}_{\bar{A}}(r) = r_5$$

where (note: relation r_i is produced by Step i)

$$r_1 = \pi_{r.V,\mathbb{P}.T_3}((r \bowtie_{C(r.T,s.T)\ \wedge\ r.\bar{G}=s.\bar{G}} \rho_s(r)) \bowtie_{r.T=\mathbb{P}.T_1 \wedge s.T=\mathbb{P}.T_2} \mathbb{P}),$$

$$r_2 = \pi_{r.V,s.T}(r \bowtie_{C(r.T,s.T)\ \wedge\ r.\bar{G}=s.\bar{G}} \rho_s(r)),$$

$$r_3 = r \cup r_1 \cup r_2, \text{ and}$$

$$r_4 =_{\bar{r}.G} F_{\bar{A}}(r_3).$$

$$r_5 = r_4 - (r_4 \ltimes_{r.\bar{G}=s.\bar{G}\ \wedge\ C(r_4.T,s.T)} \rho_s(r_4))$$

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, 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, 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.

CHAPTER 5
IMPLEMENTATION

This section describes the implementation of temporal query using log-segmented label as timestamp. In other words, here we illustrates the transformation of temporal query to a query having only operations available already in a standard SQL engines so that the transformed query gets executed in the unmodified engine which, inevitably, executes the sequence semantics in a traditional SQL engine. This section covers hardware configuration, system setup, system design and exemplification of query transformation.

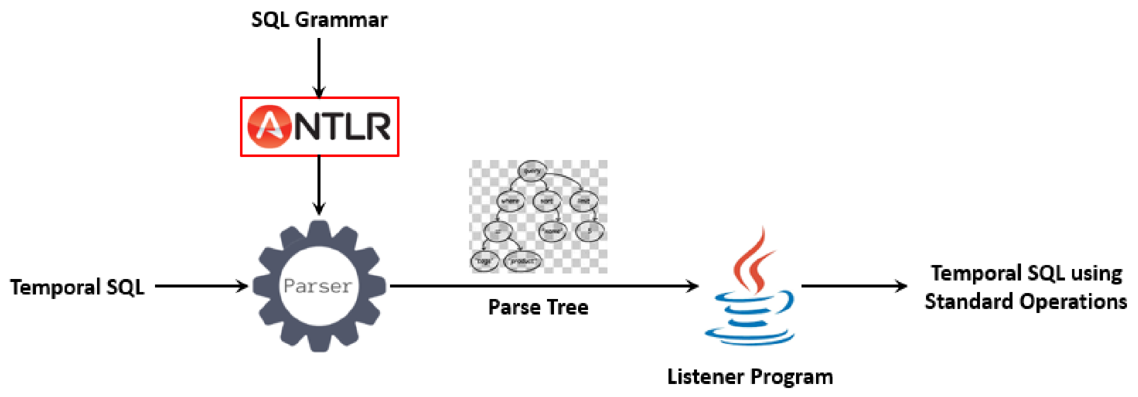

Fig. 5.1: System Design.

Host machine's hardware configuration: Intel Core i5 quad core CPU with 3MB L2 cache and 2.8 GHz clock speed, 16 GB of memory and 1 TB of hard drive. Windows 8.1 Pro 64-bit is used as host operating system.

System Setup: We have used ANTLR version 4.5.3 for implementation and testing of our research work. From a given grammar, ANTLR generates parse tree and walks through it. We have built a java program that takes the parser and a temporal query as input and generates the desired SQL consisting of only standard operations. We have used Netbeans 8.2 as java IDE and integrated ANTLR into it as a plugin. As a reference database we have chosen PostgreSQL version 11.12 database management system.

System Design: Figure 5.1 illustrates the flow of control of our implementation. The ANTLR gets a grammar (i.e., set of rules of a SQL language) and generates parser for it. The parser generates parse tree from a temporal SQL query. Then our custom listener program receives the parse tree (which enumerates the type of operation, table name(s), column name(s), join column name(s), etc.) and constructs the intended query. The key part of the implementation is the transformation of query of various SELECT operations.

We demonstrate in this section that log segmented label offers a way to implement the temporal semantics in an unaltered database. In contrast with our research, earlier articles implemented the sequence semantics by changing the SQL engine [2, 4]. This work investigates all the relational algebras of chapter 4 and deduce the transformation for each of the operations: projection, selection, Cartesian product, union, intersection, set difference and aggregation. For a subset of SQL comprised of constructive parts, like projection, selection, join operations, are comparatively straight forward using log segmented timestamps. Besides, the eliminative parts of SQL such as set difference, sequenced aggregation and grouping operations require complex query conversions. Nonetheless, with the log segmented labels, there is absolutely no need of any alternation of the underlying DBMS. The subsequent section explores the query conversion for each of the above operations with examples.

5.1 QUERY TRANSFORMATION

This section illustrates sample temporal query and their equivalent query with available techniques in a standard database (i.e., PostgreSQL). We tried to write exhaustive set of queries (to cover all cases), transformed them in order to accumulate insight on the query conversion. Finally used that knowledge to formulate the algorithm of the listener program that ultimately converts the query. The listener program consists of several subroutines and each subroutine handles a definite type of SELECT operation. The program switches to a particular subroutine based on the specific keywords in the query (e.g., "WHERE", "JOIN", "UNION", "INTERSECTION", "EXCEPT", "COUNT", "SUM", "GROUP BY" and so on) which essentially turns the listener into a rule based program. Algorithm 5.1 shows the pseudo-code of the listener. Algorithm 3.1 shows the pseudo-code of the listener. On the other hand, each subroutine takes care of one type of operation similar to the each of the relational algebra in section reference-to-the-relational-algebra-section.

Algorithm 5.1 Listener

```
Input:
    Parse Tree of a Temporal Query ($P_t$)
Output:
    Transformed Query ($Q_c$)
Begin
    if K_WHERE $\in P_t$ then handle_selection()
    else if K_JOIN $\in P_t$ then handle_Cartesian_product()
    else if K_UNION $\in P_t$ then handle_union()
    else if K_INTERSECTION $\in P_t$ then handle_intersection()
    else if K_EXCEPT $\in P_t$ then handle_difference()
    else if K_GROUP_BY $\in P_t$ then handle_aggregation()
    else handle_projection()
End
```

Even though the conversion is similar to the conversion of relational algebra operation, there is a non-trivial difference between them due to underlying data model. In contrast with the relational algebra, the SQL table allows duplicate tuples. Hence, unlike the relation algebra, there is no need of duplicate removal in conversion from sequenced to non-sequenced SQL query. We will go through each of them in the following subsections.

5.1.1 SEQUENCED PROJECTION

Figure 5.2 illustrates typical transformation of sequenced projection. Here, the query at the left hand side is the temporal query and that at the right side is the equivalent query with standard operations. The “data” column and the “projection_table” in the temporal query (highlighted with blue color) indicate the list of columns to be projected and the table name respectively. Similarly, the “data” column and the “projection_table” in the right hand side query (highlighted with the blue background) stand for the column names and the table name to be replaced with. Thus our listener program looks for the projected column list and the table name from the temporal query and then construct a query alike the right side query with the retrieved metadata. Similar to the other constructive queries, the sequenced projection is easier to transform.

```
@temporal
SELECT t1.data
FROM projection_table AS t1;
```

(a) Temporal Query

```
SELECT t1. data
     , t1.time
FROM projection_table AS t1
EXCEPT
SELECT t1. data
     , t2.time
FROM  projection_table  AS t1
JOIN  projection_table  AS t2
ON t1. data  = t2. data
WHERE t1.time != t2.time
AND t2.time LIKE t1.time || '%';
```

(b) Equivalent Query

Fig. 5.2: Transformation of Sequenced Projection

```
@temporal
SELECT t1.data
FROM selection_table AS t1
WHERE t1.data = 'data_value';
```

(a) Temporal Query

```
SELECT t1. data
     , t1.time
FROM  selection_table  AS t1
WHERE  t1.data = 'data_value' ;
```

(b) Equivalent Query

Fig. 5.3: Transformation of Sequenced Selection

5.1.2 SEQUENCED SELECTION

The conversion of sequenced selection query is pretty plain dealing. It retrieves the column list, table name and the condition of selection from the temporal query and then returns the selected columns along with the timestamp of the tuples. There is nothing to deal with time metadata apart from retrieving it. Figure 5.3 demonstrates the scenario.

```
@temporal
SELECT t1.data
     , t2.data
FROM join_table_1 AS t1
JOIN join_table_2 AS t2
ON t1.data = t2.data;
```

(a) Temporal Query

```
SELECT t1. data
     , t2. data
     , CASE
         WHEN length(t1.time) >= length(t2.time)
         THEN t1.time
         ELSE t2.time
       END AS time
FROM  join_table_1  AS t1
JOIN  join_table_2  AS t2
ON t1. data  = t2. data
WHERE t1.time LIKE t2.time || '%'
OR t2.time LIKE t1.time || '%';
```

(b) Equivalent Query

Fig. 5.4: Transformation of Sequenced Cartesian Product

5.1.3 SEQUENCED CARTESIAN PRODUCT

Unlike period timestamp, the log segmented timestamp makes the Cartesian product query conversation easier. In sequenced Cartesian product the result tuple only exists if the underlying two tuples from two tables co-exist together. Using log segmented labels, we do it by prefix checking which is simple done in SQL by a “LIKE” operation. Again, the timestamp of the result tuple will be the timestamp of the short living tuple which is simply the longer (in length) timestamp in SQL while using log segmented labels as timestamp. This is what we have did while transforming the sequenced Cartesian product as illustrated in the figure 5.4. It extracts the selected columns, join columns, join tables and replaces the template of the intended query. Additionally, it matches the prefix (of each other) and select the longer (in length) timestamp as the time of the result tuple.

5.1.4 SEQUENCED UNION

In congruence to the sequenced projection operation, the sequenced union handler captures the selected columns and union tables before replacing them into the stub query for union conversion. This is shown in the figure 5.5.

```
@temporal
SELECT t1.data
FROM union_table_1 AS t1
UNION
SELECT t2.data
FROM union_table_2 AS t2;
```

(a) Temporal Query

```
WITH union_result_set AS (
    SELECT data
         , time
    FROM union_table_1
    UNION
    SELECT data
         , time
    FROM union_table_2
)
SELECT data
     , time
FROM union_result_set
EXCEPT
SELECT t1.data
     , t2.time
FROM union_result_set AS t1
JOIN union_result_set AS t2
ON t1.data = t2.data
```

(b) Equivalent Query

Fig. 5.5: Transformation of Sequenced Union

5.1.5 SEQUENCED INTERSECTION

Sequenced intersection and sequenced Cartesian product operations are somehow homogeneous in terms of calculating timestamp of a tuple. Two value equivalent tuples from two tables only get selected in the result tuple if and only if they live in some same time period. Using log segmented labels, we determine it by prefix testing among timestamps and we select the longer (in length) timestamp as the timestamp of the tuple in order to determine the co-existence. Figure 5.6 demonstrates how we convert the query in our handler subroutine.

```
@temporal
SELECT t1.data
FROM intersection_table_1 AS t1
INTERSECT
SELECT t2.data
FROM intersection_table_2 AS t2;
```

(a) Temporal Query

```
SELECT t1. data
     , CASE
        WHEN length(t1.time) >= length(t2.time)
        THEN t1.time
        ELSE t2.time
     END AS time
FROM  intersection_table_1  AS t1
JOIN  intersection_table_2  AS t2
ON t1. data  = t2. data
WHERE t1.time LIKE t2.time || '%'
OR t2.time LIKE t1.time || '%';
```

(b) Equivalent Query

Fig. 5.6: Transformation of Sequenced Intersection

5.1.6 SEQUENCED DIFFERENCE

Sequence difference is little harder than the earlier operations. Transforming sequenced differences directly from the concept of relational algebra makes it more complex. Instead we will define some primitives, check a representative example of difference, grow some intuition behind the logic and then develop a more feasible algorithm so that calculating difference between two labels will be easier to implement in SQL. We start with the following definition.

Sibling Labels: Two log segmented labels are sibling of each other if they are of equal length and differ only in the last bit. For instance, 10110 and 10111 are siblings since they have same length and only the last bit is dissimilar with each other. But 1101 and 11010 (unequal length) or 10110 and 10101 (earlier bits are different) are not sibling. Thus if we toggle the last bit of a log segmented label then we get its sibling.

Parent Labels: If we remove the last bit of a log segmented label (apart from the root label), we get the parent of the label. Namely, 1001 is the parent label of 10010 and 10011 since removing last bit from the both of 10010 and 10011 result in 1001. By similar argument, child label forms when another bit is added at the last of a label. A parent label can have exactly two children labels which are sibling of each other. Both the sibling label and the parent label have a good link in determining set difference with log segmented timestamp which we will see by an example.

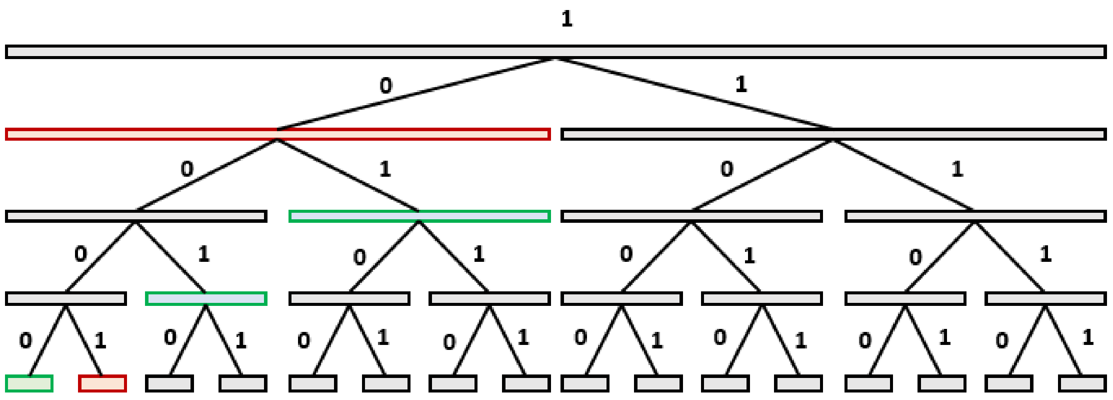

Fig. 5.7: Temporal Difference between two labels.

If we come back to the example of set difference given in the subsection 4.6, 10 – 10001 = {10000, 1001, 101}. Figure 5.7 depicts this difference where the difference between two red labels are the green labels. Careful observation of the figure reveals that,

- Result set consists of 10000 which is the sibling of short living label.
- Since two of the children labels (10000 and 10001) have been considered, the parent label (1000) need no consideration in subtraction.
- Now the sibling (1001) of the parent belongs to the result.
- The above three steps continues until one level down the long living label.

```
SELECT t1.data
 , unnest(
 string_to_array(
 (
SELECT string_agg(
(left(substring(t2.time, 1, len), len-1)::bit varying || ~right(substring(t2.time, 1, len), 1)::bit)::text
, ','
)
FROM generate_series(length(t1.time)+1, length(t2.time)) as len
 )
 ,','
 )
 ) AS TIME
FROM difference_table_1 AS t1
JOIN difference_table_2 AS t2
ON t1.data = t2.data
WHERE t1.time <> t2.time
AND t2.time LIKE t1.time || '%';
```

(a) Equivalent Query

Fig. 5.8: Transformation of Sequenced Set Difference

The above discussion sheds the light on the logic how set difference between two log segmented labels works. Last but not the least, if the long living label is not the prefix of the short living label then the operation returns “NULL”. Hence we formulate the algorithm 5.2 that calculates set difference between two log segmented labels.

The algorithm 5.2 generates a set of log segmented labels while calculating set difference between two labels. This is pretty easier to implement using SQL functions. In PostgreSQL, we construct a single row with each set element. The final query conversion looks like the figure 5.8.

5.1.7 SEQUENCED AGGREGATION AND GROUPING

From the discussion of relational algebra for sequenced grouping and aggregation (section 4.7), we found that it has a complex implementation to some extent. For ease of understanding, we first discuss our implementation approach keeping similarity with the relational algebra and then present our code conversion. In a relation, there exist two types

Algorithm 5.2 Set Difference

```
Input:
        Log segmented label ($l_1$)
        Log segmented label ($l_2$)
Output:
        Set of log segmented label(s) ($result$)
Begin
        if $l_1$ LIKE $l_2$ + '%' then
            current_label ← $l_2$
            do
                current_label ← toggle_last_bit(current_label)
                result.append(current_label)
                current_label ← remove_last_bit(current_label)
            while length(current_label) > length($l_1$)
            return $result$
        otherwise
            return $NULL$
        end if
End
```

of value equivalent tuples.

- First type of (value equivalent) tuples don't coexist with other tuple. That means, they don't live within time limit of other tuples and also don't outspan others. In this case, the grouping operation works similar to the non-temporal query.

- Second type of (value equivalent) tuples coexist with other tuples which means either their lifespan belong to that of other tuples or vice versa. In that case we have to adopt two steps.

  – We keep the timestamp of the short-lived tuple unchanged.

  – Then we split the timestamp of the long-lived tuples into multiple smaller timestamps using set difference operation.

  For instance, if we get following two value equivalent tuples with overlapping timestamps: (value1, value2, 10) and (value1, value2, 10001) then we keep the short living tuple (value1, value2, 10001) and split the long lasting tuple resulting in (value1, value2, 10000), (value1, value2, 1001), (value1, value2, 101), (value1, value2, 11) [since, $1 - 10001 = \{10000, 1001, 101\}$]. Thus we accumulate total of 5 tuples: (value1, value2, 10001), (value1, value2, 10000), (value1, value2, 1001), (value1, value2, 101), (value1, value2, 11).

Sometimes, a long living tuple outspans multiple tuples and we split the long lasting tuple with the tuples that has smallest longevity, since it produces maximum outcomes.

```
SELECT COUNT(*)
    , dept
    , time
FROM (
    WITH CTE AS (
        SELECT name_lg
            , name_sm
            , dept
            , time_lg
            , time_sm
        FROM (
            SELECT name_lg
                , name_sm
                , dept
                , time_lg
                , time_sm
                , ROW_NUMBER() OVER(PARTITION BY name_lg ORDER BY diff DESC)
            FROM (
                SELECT t1.name AS name_lg
                    , t2.name AS name_sm
                    , t1.dept AS dept
                    , t1.time AS time_lg
                    , t2.time AS time_sm
                    , length(t2.time) - length(t1.time) AS diff
               FROM aggregation_count_table AS t1
               JOIN aggregation_count_table AS t2
               ON t1.dept = t2.dept
               WHERE t1.time != t2.time AND t2.time LIKE t1.time || '%'
            ) AS inner_table
        ) AS outer_table
        WHERE row_numbr = 1
    )
SELECT name_lg AS name
    , dept
    , time_sm AS time
FROM CTE

UNION ALL

SELECT name_lg
    , dept
    , unnest(
        string_to_array(
        (
            SELECT string_agg(
                (left(substring(time_sm, 1, len), len1)::bit varying
                || ~right(substring(time_sm, 1, len), 1)::bit)::text
                , ','
            )
```

```
                FROM generate_series(length(time_lg)+1, length(time_sm)) as l
            ) AS TIME
    FROM CTE

    UNION ALL

    SELECT *
    FROM aggregation_count_table

    EXCEPT ALL

    SELECT t3.*
    FROM aggregation_count_table AS t3
    JOIN aggregation_count_table AS t4
    ON t3.dept = t4.dept -- non-temporal group by column (list)
    WHERE t3.time != t4.time
    AND t4.time LIKE t3.time || '%'
) AS t
GROUP BY dept, time;
```

5.2 TESTING

In order to test the query conversion, we have created some table and populate it with some randomly generated data. Then we have written some sequenced query and have converted them to desired query to be executed on a standard DBMS. Then we executed the transformed queries and stored the data that they returned. Simultaneously we have executed the manually transformed queries and compared the returned data with the previously stored data. At ours surprise, we got absolutely same data both times. A typical scripts of metadata and the data is given at the appendix A.

CHAPTER 6
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] M. Böhlen and C. S. Jensen, "Sequenced semantics," in *Encyclopedia of Database Systems.* Springer, 2009, pp. 2619–2621.

[2] C. E. Dyreson, V. A. Rani, and A. Shatnawi, "Unifying sequenced and non-sequenced semantics," in *2015 22nd International Symposium on Temporal Representation and Reasoning (TIME).* IEEE, 2015, pp. 38–46.

[3] M. H. Böhlen, C. Jensen, R. Snodgrass, L. Liu, and M. T. Oezsu, "Nonsequenced semantics," 2009.

[4] A. Dignös, M. H. Böhlen, and J. Gamper, "Temporal alignment," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 433–444.

[5] C. X. Chen and C. Zaniolo, "Sql st: A spatio-temporal data model and query language," in *International Conference on Conceptual Modeling.* Springer, 2000, pp. 96–111.

[6] C. E. Dyreson, "Observing transaction-time semantics with/sub tt/xpath," in *Proceedings of the Second International Conference on Web Information Systems Engineering*, vol. 1. IEEE, 2001, pp. 193–202.

[7] F. Grandi, "T-sparql: A tsql2-like temporal query language for rdf." in *ADBIS (local proceedings)*, 2010, pp. 21–30.

[8] R. Snodgrass, "The temporal query language tquel," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 2, pp. 247–298, 1987.

[9] R. T. Snodgrass, *The TSQL2 temporal query language.* Springer Science & Business Media, 2012, vol. 330.

[10] R. T. Snodgrass, Ed., *The TSQL2 Temporal Query Language.* Kluwer, 1995.

[11] J. Chomicki and D. Toman, "Abstract versus concrete temporal query languages." 2009.

[12] K. Torp, C. S. Jensen, and M. Böhlen, "Layered temporal dbms's—concepts and techniques," in *Database Systems For Advanced Applications' 97.* World Scientific, 1997, pp. 371–380.

[13] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Stratum approaches to temporal dbms implementation," in *Proceedings. IDEAS'98. International Database Engineering and Applications Symposium (Cat. No. 98EX156).* IEEE, 1998, pp. 4–13.

[14] C. Dyreson and V. A. Rani, "Translating temporal sql to nested sql," in *2016 23rd International Symposium on Temporal Representation and Reasoning (TIME).* IEEE, 2016, pp. 157–166.

[15] C. S. Jensen, C. E. Dyreson, M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer *et al.*, "The consensus glossary of temporal database concepts—february 1998 version," in *Temporal Databases: Research and Practice.* Springer, 1998, pp. 367–405.

[16] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass, "On the semantics of "now" in databases," *ACM Transactions on Database Systems (TODS)*, vol. 22, no. 2, pp. 171–214, 1997.

[17] C. A. Hurtado, A. O. Mendelzon, and A. A. Vaisman, "Maintaining data cubes under dimension updates," in *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337).* IEEE, 1999, pp. 346–355.

[18] A. A. Vaisman and A. O. Mendelzon, "A temporal query language for olap: Implementation and a case study," in *International Workshop on Database Programming Languages.* Springer, 2001, pp. 78–96.

[19] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Temporal statement modifiers," *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 4, pp. 407–456, 2000.

[20] M. Böhlen, J. Gamper, and C. S. Jensen, "Temporal aggregation," in *Encyclopedia of Database Systems.* Springer, 2009, pp. 2924–2929.

[21] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, S. Jajodia, C. Dyreson, F. Grandi, W. Käfer, N. Kline *et al.*, "A consensus glossary of temporal database concepts," *ACM Sigmod Record*, vol. 23, no. 1, pp. 52–64, 1994.

[22] A. Tansel, "Modelling temporal data," *Information and Software Technology*, vol. 32, no. 8, pp. 514–520, 1990.

[23] C. E. Dyreson, "Using couchdb to compute temporal aggregates," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* IEEE, 2016, pp. 1131–1138.

[24] C. E. Dyreson and R. T. Snodgrass, "Timestamp semantics and representation," *Information Systems*, vol. 18, no. 3, pp. 143–166, 1993.

APPENDICES

APPENDIX A

SQL Listing for Testing

```
-- Sequenced Projection
DROP TABLE IF EXISTS projection_table;
CREATE TABLE projection_table(
 data varchar(50),
 time varchar(64)
 );
INSERT INTO projection_table(data, time)
VALUES ('A', '10'),
 ('A', '101'),
 ('A', '110'),
 ('B', '11001'),
 ('B', '110'),
 ('C', '100'),
 ('C', '1001'),
 ('D', '1101'),
 ('D', '101'),
 ('E', '111'),
 ('F', '11');

-- Sequenced Selection
DROP TABLE IF EXISTS selection_table;
CREATE TABLE selection_table(
 data varchar(50),
 time varchar(64)
```

```
);
INSERT INTO selection_table(data, time)
VALUES ('A', '10'),
('A', '110'),
('B', '11001'),
('C', '100'),
('D', '1101'),
('E', '111'),
('F', '11');

-- Sequenced Cartesian Product
DROP TABLE IF EXISTS join_table_1;
CREATE TABLE join_table_1(
data varchar(50),
time varchar(64)
);
INSERT INTO join_table_1(data, time)
VALUES('A', '10'),
('B', '11001'),
('C', '100'),
('D', '1101'),
('E', '111'),
('F', '11');

DROP TABLE IF EXISTS join_table_2;
CREATE TABLE join_table_2(
data varchar(50),
time varchar(64)
```

```
);
INSERT INTO join_table_2(data, time)
VALUES('A', '10101'),
('B', '10101'),
('C', '101'),
('D', '110'),
('E', '11101'),
('F', '11');

-- Sequenced Union
DROP TABLE IF EXISTS union_table_1;
CREATE TABLE union_table_1
(
 data character varying(50),
 time character varying(64)
  );
INSERT INTO union_table_1
VALUES ('A', '11'),
('B', '1010'),
('C', '101'),
('D', '110'),
('E', '110'),
('F', '1101');
DROP TABLE IF EXISTS union_table_2;
CREATE TABLE union_table_2
(
 data character varying(50),
 time character varying(64)
```

```
);
INSERT INTO union_table_2
VALUES ('A', '1101'),
('B', '10'),
('C', '101'),
('D', '1100'),
('E', '101'),
('F', '100');

-- Sequenced Intersection
DROP TABLE IF EXISTS intersection_table_1;
CREATE TABLE intersection_table_1
(
 data character varying(50),
 time character varying(64)
);
INSERT INTO intersection_table_1
VALUES ('A', '11'),
('B', '1010'),
('C', '110'),
('D', '110'),
('E', '101'),
('F', '110');
DROP TABLE IF EXISTS intersection_table_2;
CREATE TABLE intersection_table_2
(
 data character varying(50),
 time character varying(64)
```

```
);
INSERT INTO intersection_table_2
VALUES ('A', '1101'),
('B', '10'),
('C', '101'),
('D', '100'),
('E', '101'),
('F', '1100');

-- Sequenced Difference
DROP TABLE IF EXISTS difference_table_1;
CREATE TABLE difference_table_1
(
 data character varying(50),
 time character varying(64)
);
INSERT INTO difference_table_1
VALUES ('A', '1'),
('B', '1010'),
('C', '101'),
('D', '110'),
('E', '110'),
('F', '1001');
DROP TABLE IF EXISTS difference_table_2;
CREATE TABLE difference_table_2
(
 data character varying(50),
 time character varying(64)
```

```
);
INSERT INTO difference_table_2
VALUES ('A', '1010'),
('B', '10'),
('C', '101'),
('D', '1100'),
('E', '101'),
('F', '100');

-- Sequenced Aggregation and Grouping
DROP TABLE IF EXISTS aggregation_count_table;
CREATE TABLE aggregation_count_table
(
 name character varying(50),
 dept character varying(50),
 time character varying(64)
);
INSERT INTO aggregation_count_table
VALUES ('Susan', 'Clothing', '1010'),
('Pedro', 'Clothing', '1010'),
('Malik', 'Clothing', '101'),
('Freya', 'Clothing', '1');
-- ('Fred', 'Clothing', '10'),
-- ('Joe', 'Clothing', '10101');
-- INSERT INTO aggregation_count_table
-- VALUES ('Fred', 'Clothing', '10'),
-- ('Joe', 'Clothing', '10101');
```

CURRICULUM VITAE

M A Manazir Ahsan

Conference Papers - In Press

- Dyreson, C., Ahsan, M. (2021). Achieving a Sequenced, Relational Query Language with Log-Segmented Timestamps. In 28th *International Symposium on Temporal Representation and Reasoning (TIME 2021).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik.