Utah State University
DigitalCommons@USU

All Graduate Theses and Dissertations | Graduate Studies

5-2016

Translating Temporal SQL to Nested SQL

Venkata Rani
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Rani, Venkata, "Translating Temporal SQL to Nested SQL" (2016). *All Graduate Theses and Dissertations*. 4966.
https://digitalcommons.usu.edu/etd/4966

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

TRANSLATING TEMPORAL SQL TO NESTED SQL

by

Venkata Rani

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

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Curtis Dyreson Major Professor	Dr.Nicholas Flann Committee Member
Dr. Kyumin Lee Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2016

Copyright © Venkata Rani 2016

All Rights Reserved

ABSTRACT

Translating Temporal SQL to Nested SQL

by

Venkata Rani, Master of Science

Utah State University, 2016

Major Professor: Dr. Curtis Dyreson
Department: Computer Science

Sequenced and *nonsequenced* semantics are the two previously researched semantics for the evaluation of an operation in a temporal database such as a query or data modification. Sequenced semantics evaluates an operation in each time instant using only the data alive at that time. Nonsequenced semantics, in contrast, means that an operation explicitly references and manipulates the timestamps in the data.

In this thesis we propose a new framework that shows both semantics are variants of a general *temporal semantics*. We present the general semantics and show how additional semantics, such as *preceding* semantics can be realized. The semantics are specified using annotations.

The primary contribution of this thesis is the translation from *temporal SQL* to *nested SQL*. We focus on SQL's SELECT statement, which is used to query data. Temporal SQL is SQL annotated with temporal semantics. Nested SQL is SQL for non-1NF data, with additional operations, such as COGROUP and FLATTEN to create and un-nest, respectively, bags of tuples (non-1NF data). This thesis develops a denotational semantics for translating from temporal to nested SQL. We implemented the denotational semantics for an SQLite ANTLR grammar, and the thesis also reports on the implementation.

(53 pages)

PUBLIC ABSTRACT

Translating Temporal SQL to Nested SQL

Venkata Rani

Sequenced and *nonsequenced* semantics are the two previously researched semantics for the evaluation of an operation in a temporal database such as a query or data modification. Sequenced semantics evaluates an operation in each time instant using only the data alive at that time. Nonsequenced semantics, in contrast, means that an operation explicitly references and manipulates the timestamps in the data.

In this thesis we propose a new framework that shows both semantics are variants of a general *temporal semantics*. We present the general semantics and show how additional semantics, such as *preceding* semantics can be realized. The semantics are specified using annotations.

The primary contribution of this thesis is the translation from *temporal SQL* to *nested SQL*. We focus on SQL's SELECT statement, which is used to query data. Temporal SQL is SQL annotated with temporal semantics. Nested SQL is SQL for non-1NF data, with additional operations, such as COGROUP and FLATTEN to create and un-nest, respectively, bags of tuples (non-1NF data). This thesis develops a denotational semantics for translating from temporal to nested SQL. We implemented the denotational semantics for an SQLite ANTLR grammar, and the thesis also reports on the implementation.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Curtis Dyreson for supporting me throughout this project. His belief in my abilities and support helped me stay on course and work towards my goal.

I would also like to take this opportunity to thank Dr. Nicholas Flann and Dr. Kyumin Lee for being on my committee and for providing me with their invaluable feedback and suggestions.

Last but not the least, would like to thank my family and friends who stood by me through thick and thin.

Venkata Rani

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . v

ACKNOWLEDGMENTS . . . vi

LIST OF FIGURES . . . ix

1 INTRODUCTION . . . 1

1.1 Related Work . . . 3

2 BACKGROUND . . . 4

2.1 Preliminaries . . . 4

2.2 Lineage . . . 5

2.3 Nested SQL Operations and Timestamp Operations . . . 6

2.3.1 COGROUP . . . 6

2.3.2 FLATTEN . . . 7

2.3.3 Timestamp Operations . . . 7

3 SEMANTICS . . . 11

3.1 Sequenced Semantics . . . 11

3.2 (Implicit) Nonsequenced Semantics . . . 13

3.3 General Temporal Semantics . . . 15

3.4 Preceding Semantics . . . 16

3.5 Context Semantics . . . 17

3.6 Distinguishing Transitions . . . 19

3.7 Precise Semantics . . . 20

4 ANNOTATIONS 22

4.1 Overview of Annotations 22

4.2 Formal Specificatio of the Annotation Language 24

4.2.1 Annotation Examples 26

4.3 Precise Manipulation of Timestamps 28

4.3.1 Sequenced Support is Key 29

5 TRANSLATION 30

5.1 Translation Example 30

5.2 A Denotational Semantics for Temporal SQL 31

5.2.1 INTERSECT 33

5.2.2 EXCEPT 33

5.2.3 UNION 34

5.2.4 Subqueries 35

5.2.5 Outer Join 38

5.2.6 Grouping and Aggregation 39

5.3 Examples 39

6 IMPLEMENTATION 43

6.1 Translation using SQLite and ANTLR 44

7 CONCLUSION AND FUTURE WORK 48

LIST OF FIGURES

Figure Page

1.1 Snapshot reducibility of sequenced semantics 2

1.2 Nonsequenced reduces to non-temporal semantics 2

2.1 Employee and Depatment 5

2.2 Join Attribute of Employee, Department 7

2.3 Join Vs CoGroup 8

2.4 Timestamp operations 9

2.5 Coalescing 10

3.1 Sequenced semantics 12

3.2 Sequenced results 14

3.3 Merging the maximum salary using lineage 15

3.4 Nonsequenced semantics 15

3.5 Nonsequenced results 16

3.6 Preceding semantics 17

3.7 Preceding semantics results 18

3.8 Preceding semantics within a context of two chronon periods 19

5.1 Example Tables of Employees . . . 40

5.2 After COGROUP . . . 40

5.3 After COALESCE . . . 40

5.4 After INTERSECT . . . 40

5.5 After FLATTEN . . . 41

5.6 Preceding Except Tables . . . 41

5.7 Preceding COGROUP . . . 41

5.8 Preceding COALESCE . . . 42

5.9 Preceding EXCEPT Result . . . 42

5.10 Nonsequenced Interpretation . . . 42

5.11 Nonsequenced COGROUP . . . 42

5.12 Nonsequenced FLATTEN . . . 42

CHAPTER 1

INTRODUCTION

Sequenced and *nonsequenced* semantics were introduced as different semantics for the evaluation of a temporal operation such as a query or data modification. Böhlen and Jensen trace the history and meaning of sequenced semantics [1], but, put simply, sequenced semantics evaluates an operation in each time instant using only the data alive at that time. Nonsequenced semantics, in contrast, means that an operation explicitly references and manipulates the timestamps in the data [2]. In some sense, nonsequenced semantics is the absence of a implicit temporal semantics, only explicit, direct manipulation of the timestamps is supported.

One important benefit of both semantics is that they *reduce* to non-temporal semantics. For sequenced semantics, the reducibility is called *snapshot reducibility* [3] or *S-reducibility* [4]. The idea is sketched in Figure 1.1. In the figure, temporal data is data annotated (in some fashion) with times. The meaning of the sequenced evaluation of a query on the temporal data is that the result has to be *slice* or *snapshot equivalent* [5] to evaluating the query using non-temporal semantics on each slice of the data. So the temporal semantics is defined in terms of a (presumably easily understood) *slice* of temporal to non-temporal, and the non-temporal semantics of query evaluation (also, well understood).

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

Traditionally, the two semantics have been seen as profoundly different. In this thesis (and previously published paper [6]) we reconcile the differences. We make the following contributions.

- We show that sequenced and nonsequenced semantics are variants of a more general temporal semantics. We describe the general semantics and show additional semantics, such as

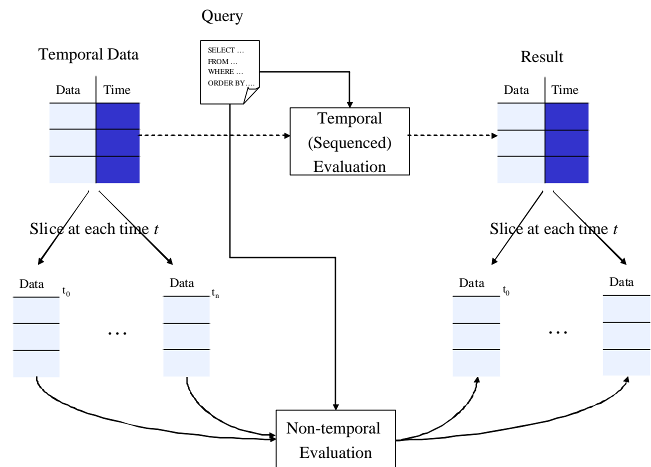

Fig. 1.1: Snapshot reducibility of sequenced semantics

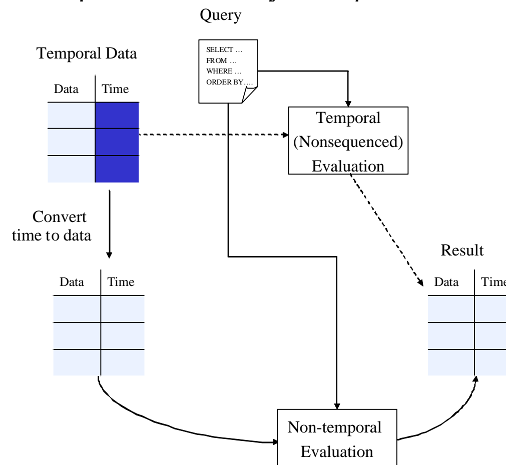

Fig. 1.2: Nonsequenced reduces to non-temporal semantics

preceding semantics.

- We show how each semantics can be specified in a query language using *annotations*, which are similar to statement modifiers, but more lightweight and syntactically separate from each query expressed in the language.

- We provide a *denotational semantics* for translating *temporal SQL* to *nested SQL*. Temporal SQL is SQL annotated with a temporal semantics. Nested SQL is SQL for a non-1NF data model. The denotational semantics is the core contribution of this thesis.

- We describe an implementation of the denotational semantics for the SELECT statement in an SQLite ANTLR grammar.

1.1 Related Work

This thesis extends previous research in the area of temporal query languages, more specifically it investigates what it means to make a query *temporal*. There are many temporal extensions of query languages, *c.f.,* [3, 7–12]. 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. We broaden the meaning of sequenced semantics in this thesis to cover a wide variety of implicit manipulation. 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 [13]. Time is just one kind of metadata, so languages that support a temporal extension can also been extended to cover other kinds of metadata using aspect-oriented techniques [14, 15].

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 [16, 17]. Some constructs prove difficult to translate, temporal outer join for instance, so a second approach is to extend the DBMS itself [18]. Since temporal semantics are (largely) reductive, both approaches at their core reuse SQL. But 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. In this thesis we adopt the first option, but translate to nested SQL rather than SQL. "Nested SQL" is something we invented by adding what we need to SQL. We feel that the second approach makes it clearer at the language level which extensions to SQL are needed for temporal semantics in general, and as a blueprint for needed extensions to support other kinds of metadata, such as privacy, lineage, provenance, etc.

CHAPTER 2

BACKGROUND

In this chapter we review background concepts and material.

2.1 Preliminaries

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

- We use a discrete time-line, with chronons ranging from time $-\infty$ to time ∞.
- There is only one time dimension.
- We extend SQL's data model (relations are bags of tuples) to a temporal data model in which every tuple in every relation is annotated with *temporal metadata* that records the lifetime of the tuple in some time dimension. That is, it is a *tuple-timestamped* model [19].
- A tuple's lifetime is a temporal period, *e.g.,* $[b, e]$ represents the time from b to e, inclusive.

As a running example, consider the `employees` and `departments` relations depicted in Figure 2.1. The *Metadata* columns record the metadata annotations for each tuple (the lineage metadata will be explained below). For instance, the employee `Joe` worked in the `Shoes` department earning a salary of `40000` from time `1` through `7`.

The **slice** (or **snapshot**) operation produces the non-temporal state for a given time period.

Definition [Slice] Let R be a temporal relation (that is, every tuple in the relation is annotated with temporal metadata), $\mathbf{data}(s)$ be a function that strips a tuple s of its temporal metadata, $\mathbf{time}(s)$ be a function that yields the temporal metadata for a tuple s, and $[b, e]$ be a temporal period. Then

$$\mathbf{slice}(R, t, [b, e]) = (\{\ \mathbf{data}(s) \mid s \in R\ \wedge\ \mathbf{time}(s)\ \cap\ [b, e] \neq \emptyset\ \}, t)$$

Data			*Metadata*	
name	**salary**	**dept**	**time**	**lineage**
Joe	40000	Shoes	[1,7]	{a}
Joe	41000	Hats	[8,9]	{b}
Fred	42000	Shoes	[6,9]	{c}
Mary	20000	Shoes	[1,2]	{d}
Mary	62000	Camera	[8,9]	{e}

(a) The `employees` relation

Data		*Metadata*	
dept	**floor**	**time**	**lineage**
Shoes	4	[3,5]	{f}
Shoes	2	[6,9]	{g}
Camera	3	[2,9]	{h}

(b) The `departments` relation

Fig. 2.1: Employee and Depatment

Note that the sliced relation, which had been stripped of its temporal metadata, is annotated or associated with the slice time, t; t is usually in $[b, e]$, but not always. The slice of a temporal database (a set of relations) is the slice of each relation in the database. □

2.2 Lineage

Lineage metadata keeps track of which tuples are used to produce a result tuple, *c.f.,* [18,20,21]. Lineage for temporal relational algebra is described in detail elsewhere [18]. To record the lineage, each tuple, s, in each relation is assigned a globally-unique identifier, $\mathbf{id}(s)$. The lineage of a tuple in the result is the set of identifiers corresponding to each tuple used to produce the result. For example, suppose that a projection (without duplicate elimination) on column A is applied to tuple s. Then the lineage of the result is $\{\mathbf{id}(s)\}$.

Other common query operations (in SQL) are grouping and duplicate elimination (DISTINCT). The lineage of a *group* is the union of the lineage of each tuple in the group, while the lineage of the result of a duplicate elimination is also a group, the group consisting of the lineage of each duplicate.

Some operations, such as join and Cartesian product, involve *combinations* of tuples from two

or more relations. The lineage tracks the combinations that produce a tuple. For example, suppose that tuple s joins with tuple w to produce tuple r in the result, then the lineage of r includes the tuple $(\mathbf{id}(s), \mathbf{id}(w))$ denoting that this combination produced a result.

Let the *lineage evaluation* of a query, Q, on a (non-temporal) database state, S, be denoted $\mathbf{eval}(Q, S)$.

2.3 Nested SQL Operations and Timestamp Operations

This thesis proposes translating from Temporal SQL to nested SQL. Nested SQL is SQL for non-1NF relations. A relation is in first normal form (1NF) if the domain of each attribute contains only atomic (indivisible) values, and the value of each attribute contains only a single value from that domain. Non-1NF domains include bags and sets. Our temporal semantics requires some operations that produce non-1NF relations (note that these operations are present in other database query languages, such as Pig Latin [22]).

SQL lacks operations on bags or sets of tuples and timestamps. In this chapter we describe the nested SQL operations that we need. We also present additional timestamp operations on bags of timestamps.

2.3.1 COGROUP

COGROUP is similar to an inner join in SQL, in that both relate tuples from a pair of relations. The difference is that an inner join creates a 1NF relation containing the joined result, while COGROUP forms bags of tuples that would join in an inner join grouped by their their common field(s) (the fields that they would join on). The result of a COGROUP is a relation of three parts. The first part is the join attributes. The second part is a bag of tuples of the first relation that have the same values as the join attributes in the first part. The third part is a bag of tuples with the same values of the join attributes from the second relation. The Cartesian product of the two bags forms the tuples in the join.

To illustrate the difference consider the two relations in Figure 2.2. The (inner) JOIN and COGROUP of the two relations is shown in Figure 2.3. The COGROUP is the bags of tuples that would join if the relations actually were joined.

name	salary	dept
Joe	40000	Shoes
Joe	41000	Hats
Fred	42000	Shoes
Mary	20000	Shoes
Mary	62000	Camera

(a) The `employees` relation

dept	floor
Shoes	4
Shoes	2
Camera	3

(b) The `departments` relation

Fig. 2.2: Join Attribute of Employee, Department

2.3.2 FLATTEN

FLATTEN eliminates a level of nesting. Given a tuple which contains a bag, FLATTEN will emit several tuples each of which contains one tuple from the bag. To illustrate FLATTEN, consider the co-grouped relation in Figure 2.3(b). If the relation is FLATTENed, it will produce the relation in Figure 2.3(a).

2.3.3 Timestamp Operations

The functions in this section take one or two bags of timestamps, and produce a bag of tuples.

TS_INTERSECTION

TS_INTERSECTION performs bag intersection on two bags of (period) timestamps.

Definition [TS_INTERSECTION] Let b_1 and b_2 be two bags of timestamps.

$$\text{TS_INTERSECTION}(b_1, b_2) = \{\ t_1 \bigcap t_2 \mid t_1 \in b_1 \ \wedge\ t_2 \in b_2\ \}$$

Consider the bag of timestamps in Figure 2.4(a) and Figure 2.4(b). Their bag intersection is shown in Figure 2.4(c). `[1,7]` is in the result because `[1,7]` intersects `[1,14]` at times `[1,7]`. Similarly, `[6,6]` is in the result because `[1,6]` intersects `[6,9]` only at time `[6,6]`.

name	salary	dept	floor
Joe	40000	Shoes	4
Joe	40000	Shoes	2
Fred	42000	Shoes	4
Fred	42000	Shoes	2
Mary	20000	Shoes	4
Mary	20000	Shoes	2
Mary	62000	Camera	3

(a) The result of `employees JOIN departments`

dept	employees	departments
Shoes	{(Joe, 40000, Shoes), (Fred, 42000, Shoes), (Mary, 20000, Shoes)}	{(Shoes, 4), (Shoes, 2)}
Camera	{(Mary, 62000, Camera)	{(Camera, 3)}

(b) The result of `employees COGROUP departments`

Fig. 2.3: Join Vs CoGroup

TS_UNION

TS_UNION performs bag union on two bags of (period) timestamps.

Definition [TS_UNION] Let b_1 and b_2 be two bags of timestamps.

TS_INTERSECTION(b_1, b_2) = $\{ \ t_1 \bigcup t_2 \mid t_1 \in b_1 \ \wedge \ t_2 \in b_2 \ \}$

Consider the bag of timestamps in Figure 2.4(a) and Figure 2.4(b). Their bag union is shown in Figure 2.4(d).

TS_DIFFERENCE

TS_DIFFERENCE performs bag difference on two bags of (period) timestamps.

Definition [TS_difference] Let b_1 and b_2 be two bags of timestamps.

TS_INTERSECTION(b_1, b_2) = $\{ \ t_1 - t_2 \mid t_1 \in b_1 \ \wedge \ t_2 \in b_2 \ \wedge \ \not(t_1 \subset t_2) \ \}$

Consider the bag of timestamps in Figure 2.4(a) and Figure 2.4(b). Their bag difference is shown in Figure 2.4(e).

timestamp
`[1,7]`
`[8,9]`
`[6,9]`

(a) Timestamps in a bag of employees

timestamp
`[1,14]`
`[1,6]`
`[2,5]`

(b) Timestamps in another bag of employees

timestamp
`[1,7]`
`[8,9]`
`[6,9]`
`[1,6]`
`[6,6]`
`[2,5]`

(c) The bag intersection, `TS_INTERSECTION`

timestamp
`[1,7]`
`[8,9]`
`[6,9]`
`[1,14]`
`[1,6]`
`[2,5]`

(d) The bag union, `TS_UNION`

timestamp
`[6,7]`
`[8,9]`
`[7,9]`
`[1,1]`
`[6,7]`
`[8,9]`
`[6,9]`

(e) The bag difference, `TS_DIFFERENCE`

Fig. 2.4: Timestamp operations

TS_COALESCE

The TS_COALESCE operation is unary and takes as input a bag of timestamps and returns a maximal set of disjoint, non-overlapping timestamps. Essentially, TS_COALESCE is the union of the timestamps with duplicate elimination and merging of adjacent timestamps. Consider for instance the bag of timestamps in Figure 2.4(a). When coalesced, the bag becomes the bag of timestamps (there is only one timestamp in the bag) shown in Figure 2.5(a).

timestamp
[1,9]

(a) Coalescing of the bag in Figure 2.4(a)

timestamp
[1,14]

(b) Coalescing of the bag in Figure 2.4(b)

Fig. 2.5: Coalescing

CHAPTER 3

SEMANTICS

In this chapter we review sequenced and nonsequenced semantics and offer a variety of additional semantics.

3.1 Sequenced Semantics

Sequenced semantics is perhaps the most straightforward temporal semantics. To understand sequenced semantics, we can imagine the history of a database as a sequence of *states*, as depicted in Figure 3.1. The state at time t consists of the (non-temporal) relations and data in the database at time t, annotated with the metadata t.

Sequenced semantics for query evaluation stipulates that a query is logically evaluated independently on every database state. In Figure 3.1, the dashed box that surrounds the state at time t shows the state on which the query is evaluated.

Definition [Sequenced query evaluation] The sequenced evaluation of a query, Q, on a temporal database, D, is

$$\begin{aligned} \mathbf{sqEval}(Q, D) = \mathbf{merge}(\ \{\ (R, t) \mid \\ & R = \mathbf{eval}(Q, S) \\ & \wedge -\infty \leq t \leq \infty \\ & \wedge\ \mathbf{slice}(D, t, [t, t]) = (S, t)\ \}\ ) \end{aligned}$$

□

Note that sequenced query evaluation is defined in the context of a **merge** operation, which we define next.

Sequenced query evaluation (potentially) creates many duplicates. If the states at time t and $t+1$ are identical then so are the query results, *i.e.,* $R_t = R_{t+1}$. To reduce duplication in the results

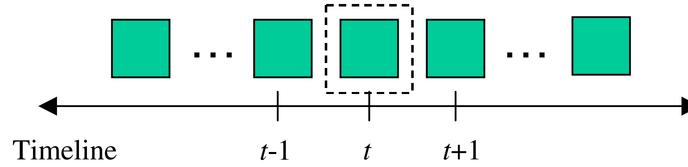

Fig. 3.1: Sequenced semantics

and associate a time with each result tuple, the results can be *merged* or *coalesced using lineage*. Coalescing without lineage has been covered elsewhere [23–25].

Keeping track of lineage prevents value-equivalent duplicates originally present in the data from being coalesced [18]. *Merge* can be defined as follows.

Definition [Merge] Let

- D be a temporal database,
- $L(s)$ be the lineage of tuple s, and
- $\bar{X} = \{(W, t) \mid -\infty \leq t \leq \infty\}$ be a set of non-temporal (but with lineage) relations, where each relation is associated with the time t (as might be produced by **slice**),

then

$$\begin{aligned}\mathbf{merge}(\bar{X}) = \{(r, [b, e]) \mid \\ & \forall t\ [\ b \leq t \leq e \ \Rightarrow\ ((W, t) \in \bar{X} \ \wedge\ (r, L(r)) \in W)\ ] \\ & \wedge ((Y, b-1) \in \bar{X} \ \Rightarrow\ (r, L(r)) \notin Y) \\ & \wedge ((Z, e+1) \in \bar{X} \ \Rightarrow\ (r, L(r)) \notin Z) \\ & \}.\end{aligned}$$

□

Consider three examples of sequenced query evaluation using the relations given in Figure 2.1. The first query projects the departments of employees that earn more than 40000.

```
SELECT dept
FROM employees
WHERE salary >= 40000 AND dept = "Shoes"
```

The result of the sequenced evaluation of the query is depicted in Figure 3.2(a). A projection in SQL preserves duplicates.

The second query finds the maximum salary of the employees.

```
SELECT MAX(salary)
FROM employees
```

The result of the sequenced evaluation of the query is depicted in Figure 3.2(b). The maximum at each time instant varies since at times different employees earned the maximum salary. Figure 3.3 depicts the merge for the maximum salary. The tuples in the `employees` relation are depicted on the timeline. If the tuple is in the slice at a given time, it is represented with a circle. A filled in circle represents the tuple with the maximum salary in that slice. The dashed boxes are the slices that are merged. Each merged set of slices is a maximum coalescing of the lineage, *i.e.,* those tuples that went into producing the result (the group that was alive in the slice).

The third query performs a join.

```
SELECT name, dept, floor
FROM employees JOIN departments ON (dept)
WHERE employees.dept = "Shoes"
```

The result of the sequenced evaluation of the query is depicted in Figure 3.2(c). An employee may have a lifetime that spans working in multiple departments, but the join limits the lifetime to when the employee worked in a given department.

3.2 (Implicit) Nonsequenced Semantics

Nonsequenced semantics has traditionally been viewed as quite different than sequenced semantics, but nonsequenced semantics is really just a different way to understand a database state at time t. Implicitly, in nonsequenced semantics a state at time t is the database slice from time $-\infty$ to ∞, that is, the database's entire history. So the nonsequenced state at time t is every relation and all of the data in every relation over the entire history of the database.

We show nonsequenced semantics in Figure 3.4. In the figure, the dashed box encloses the data on which the query is evaluated at time t, which is the entire history of the database. Except

Data	*Metadata*	
dept	**time**	**lineage**
Shoes	[1,7]	{a}
Shoes	[6,9]	{c}

(a) Sequenced projection result

Data	*Metadata*	
max	**time**	**lineage**
40000	[1,2]	{a,d}
40000	[3,5]	{a}
42000	[6,7]	{a,c}
62000	[8,9]	{b,e}

(b) Sequenced aggregate result

Data			*Metadata*	
name	**dept**	**floor**	**time**	**lineage**
Joe	Shoes	4	[3,5]	{(a,f)}
Fred	Shoes	4	[6,7]	{(c,f)}
Fred	Shoes	2	[7,9]	{(c,g)}

(c) Sequenced join result

Fig. 3.2: Sequenced results

for the difference in state, nonsequenced query evaluation is exactly the same as sequenced query evaluation.

Definition [Nonsequenced evaluation] The nonsequenced evaluation of a query, Q, on a temporal database, D, is

$$\begin{aligned}\mathbf{nsqEval}(Q, D) = \mathbf{merge}(\{ (R, t) \mid & \\ & R = \mathbf{eval}(Q, S) \\ & \wedge -\infty \leq t \leq \infty \\ & \wedge\ \mathbf{slice}(D, t, [-\infty, \infty]) = (S, t) \})\end{aligned}$$

□

Note that the **eval** produces identical results for every time, that is for all t, $R_t = R_{t+1}$. Hence, every tuple in the result lives at every time t, so the merge produces the (not very useful) timestamp

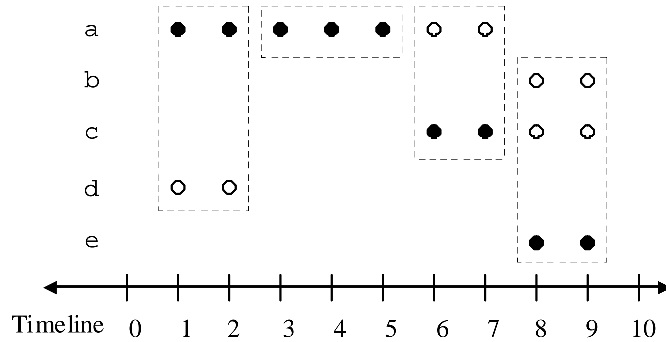

Fig. 3.3: Merging the maximum salary using lineage

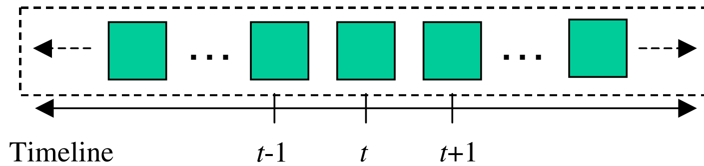

Fig. 3.4: Nonsequenced semantics

$[-\infty, \infty]$ for every result tuple. Since the timestamp is not very useful, explicit manipulation of the timestamp is the standard indicator of nonsequenced semantics.

Figure 3.5 shows the result of (implicit) nonsequenced query evaluation on the three example queries using the relations given in Figure 2.1. One difference with the sequenced result is for the query to find the maximum salary. Unlike the sequenced evaluation, the query finds the maximum across the entire history. The result of the nonsequenced evaluation of the join is given in Figure 3.5(c). Employees are related to every department at which they worked in their entire history.

3.3 General Temporal Semantics

The only difference between sequenced and nonsequenced is in how the data is sliced, which lets us generalize the two semantics.

Definition [Temporal semantics for query evaluation] The temporal evaluation of a query, Q, with a given slice function, $\mathcal{F}$, is

$$\mathbf{timeEval}(Q, \mathcal{F}) = \mathbf{merge}(\ \{\ (R, t) \mid$$

$$R = \mathbf{eval}(Q, S)$$

Data dept	*Metadata* time	lineage
Shoes	$[-\infty,\infty]$	{a}
Shoes	$[-\infty,\infty]$	{c}

(a) Nonsequenced projection result

Data max	*Metadata* time	lineage
62000	$[-\infty,\infty]$	{a,b,c,d,e}

(b) Nonsequenced aggregate result

name	*Data* dept	floor	*Metadata* time	lineage
Joe	Shoes	4	$[-\infty,\infty]$	{(a,f)}
Joe	Shoes	2	$[-\infty,\infty]$	{(a,g)}
Fred	Shoes	4	$[-\infty,\infty]$	{(c,f)}
Fred	Shoes	2	$[-\infty,\infty]$	{(c,g)}
Mary	Shoes	4	$[-\infty,\infty]$	{(d,f)}
Mary	Shoes	2	$[-\infty,\infty]$	{(d,g)}
Mary	Camera	3	$[-\infty,\infty]$	{(e,h)}

(c) Nonsequenced join result

Fig. 3.5: Nonsequenced results

$$\wedge -\infty \leq t \leq \infty$$

$$\wedge\ \mathcal{F}(t) = (S, t)\ \}\ )$$

□

Sequenced and nonsequenced semantics can be expressed as follows.

- $\mathbf{sqEval}(Q, D) = \mathbf{timeEval}(Q, \mathbf{slice}(D, t, [t, t]))$
- $\mathbf{nsqEval}(Q, D) = \mathbf{timeEval}(Q, \mathbf{slice}(D, t, [-\infty, \infty]))$

3.4 Preceding Semantics

With our new understanding of the "state" of a database, we can articulate other semantics of interest. Suppose that we define the state at time t to consist of all of the data up to and including time t as shown in Figure 3.6. We can define a new semantics, which we call *preceding semantics*

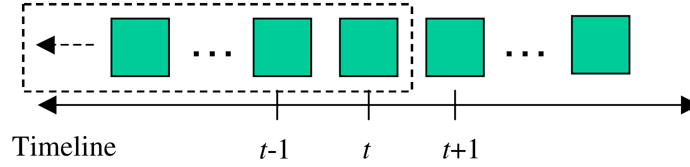

Fig. 3.6: Preceding semantics

that evaluates a query with respect to all of the data in the database up to the given time. Let $\mathcal{H}(t) = \mathbf{slice}(D, t, [-\infty, t])$ then

$$\mathbf{preceedingEval}(Q, D) = \mathbf{timeEval}(Q, \mathcal{H}).$$

The semantics of query evaluation remains the same, only the meaning of the state differs. Figure 3.7 gives the results of evaluating the three queries using preceding semantics on the relations given in Figure 2.1. The result of the preceding evaluation of the query to find the maximum employee salary is depicted in Figure 3.7(b). Unlike the sequenced evaluation, the query finds the maximum up to the time t, *i.e.,* what is the maximum to this point in time.

3.5 Context Semantics

Nonsequenced and preceding semantics both evaluate a query with respect to the start of the history of a database. But in many queries, it is beneficial to use a more restricted window. Hence, we can stipulate a *context* in which a query is evaluated. The context can be a fixed window, *e.g.,* 2012, or a periodic partitioning, *e.g.,* yearly.

Context is just a different slice function in the general temporal semantics. For example, suppose we want to use sequenced semantics in the context of the period $[3, 7]$. Let

$$\mathcal{C}(t) = \ \mathbf{if}\ 3 \leq t \leq 7\ \mathbf{slice}(D, t, [t, t])\ \mathbf{else}\ \emptyset$$

then $\mathbf{timeEval}(Q, \mathcal{C})$ uses sequenced evaluation in the period from 3 to 7. Alternatively, suppose we want implicit nonsequenced semantics in the context $[3, 7]$. Let

$$\mathcal{G}(t) = \ \mathbf{if}\ 3 \leq t \leq 7\ \mathbf{slice}(D, t, [3, 7])\ \mathbf{else}\ \emptyset$$

Data	*Metadata*	
dept	**time**	**lineage**
Shoes	[1,∞]	{a}
Shoes	[6,∞]	{c}

(a) Preceding projection result

Data	*Metadata*	
max	**time**	**lineage**
40000	[1,5]	{a,d}
42000	[6,7]	{a,c,d}
62000	[8,∞]	{a,b,c,d,e}

(b) Preceding aggregate result

	Data		*Metadata*	
name	**dept**	**floor**	**time**	**lineage**
Joe	Shoes	4	[3,∞]	{(a,f)}
Joe	Shoes	2	[6,∞]	{(a,g)}
Fred	Shoes	4	[6,∞]	{(c,f)}
Fred	Shoes	2	[6,∞]	{(c,g)}
Mary	Shoes	4	[3,∞]	{(d,f)}
Mary	Shoes	2	[6,∞]	{(d,g)}
Mary	Camera	3	[8,∞]	{(e,h)}

(c) Preceding join result

Fig. 3.7: Preceding semantics results

then $\mathbf{timeEval}(Q, \mathcal{G})$ uses only the tuples in $[3, 7]$.

Periodic slicing can also be specified. Let

$$\mathcal{P}(t) = \mathbf{slice}(D, t, [\lfloor t/2 \rfloor * 2, t])$$

then $\mathbf{timeEval}(Q, \mathcal{P})$ stipulates preceding semantics within a periodic window of size 2 (every 2 chronons). Figure 3.8 shows the result of evaluating the example queries using this semantics. Note that unlike the preceding semantics the slice window at time t only extends back in time to (at most) $t - 1$.

3.6 Distinguishing Transitions

To this point, the slice function has not depended on the timestamps being sliced. But slice functions can be defined to slice relative to (combinations of) timestamps. This is important in distinguishing transitions in the data. That is, a semantics may take interest in whether a tuple at time t, *began* its lifetime, *ended* its lifetime, or *continued* its lifetime. Such distinctions are often of interest to a user. For instance, suppose that a user wants to know "which employees started working in Shoes in [3,7]?" Distinguishing transitions of tuples into and out of states provides a semantics for evaluating such queries. Let $\mathbf{start}(R, t, [b, e])$ represent the collection of tuples that

Data	*Metadata*	
dept	**time**	**lineage**
Shoes	[1,8]	{a}
Shoes	[6,12]	{b}

(b) Preceding context projection result

Data	*Metadata*	
max	**time**	**lineage**
40000	[1,5]	{a,d}
42000	[6,7]	{a,c,d}
62000	[8,12]	{a,b,c,d,e}

(b) Preceding context aggregate result

	Data		*Metadata*	
name	**dept**	**floor**	**time**	**lineage**
Joe	Shoes	4	[3,8]	{(a,f)}
Joe	Shoes	2	[6,8]	{(a,g)}
Fred	Shoes	4	[6,12]	{(c,f)}
Fred	Shoes	2	[6,12]	{(c,g)}
Mary	Shoes	4	[3,4]	{(d,f)}
Mary	Camera	3	[8,12]	{(e,h)}

(c) Preceding context join result

Fig. 3.8: Preceding semantics within a context of two chronon periods

began a lifetime in $\mathbf{slice}(R, t, [b, e])$, $\mathbf{finish}(R, t, [b, e])$ represent the collection of tuples that ended a lifetime at time t in database R, and $\mathbf{continue}(R, t, [b, e])$ represent the collection of tuples that

continued (neither started nor ended) a lifetime at time t in database R. Note that $\mathbf{slice}(R, t, [b, e])$ equals

$$\mathbf{start}(R, t, [b, e]) \cup \mathbf{finish}(R, t, [b, e]) \cup \mathbf{continue}(R, t, [b, e])$$

Start, finish, and continue can be defined using slicing. For instance start is the following slice function, with b refering to a timestamp's begin time (note that the slice is applied to a tuple to determine membership in the slice, so the tuple's timestamp is available).

$$\mathcal{S}(t) = \textbf{ if } t == b \, \mathbf{slice}(D, t, [t, t]) \textbf{ else } \emptyset$$

To determine which employees started working in Shoes from [3-7], let

$$\mathcal{B}(t) = \textbf{ if } 3 \leq t \leq 7 \, \mathbf{start}(D, t, [t, t]) \textbf{ else } \emptyset,$$

then we would evaluate with the semantics $\mathbf{timeEval}(Q, \mathcal{B})$ (assuming the data captures the starting times in the tuples).

3.7 Precise Semantics

Continuing with the theme of using timestamps implicit in the data, consider a semantics that slices based on the whether a pair of timestamps meets. That is, let t_1 and t_2 represent a pair of timestamps, as might be involved in a combination of tuples in a join or Cartesian product. Let

$$\mathcal{P}(t) = \textbf{ if } t_1.e == t_2.s \, \mathbf{start}(D, t_2.s, [t_2.s, t_2.s]) \textbf{ else } \emptyset,$$

where $t.s$ and $t.e$ select the begin and end times of a timestamp, respectively. Then we would evaluate a precise (meets) semantics with $\mathbf{timeEval}(Q, \mathcal{P})$.

The semantics might be used for instance to find which employees got a raise (using an inequality join) or from the employees that got a raise, who earned the maximum salary for the raised salary (and when).

These are only a few of the many possible varieties of semantics that can be expressed. We focus next on how the semantics can be woven into a query using an annotation.

CHAPTER 4

ANNOTATIONS

In this chapter we show how to specify the semantics in a language, like SQL, using annotations. Annotations are a lightweight language extension, often used to signify meta-actions. For instance annotations are used in Java for documentation (*e.g.,* a `@param` annotation to document a parameter) or extensibility (*e.g.,* a `@key` to specify a key attribute in BerkeleyDB-java).

This chapter also discusses how lineage can be used to implement a semantics. Recall that lineage tracks which tuples lead to the production of a result tuple.

4.1 Overview of Annotations

Syntactically, a temporal annotation is a specification of how to construct a timestamp from a list of timestamps. Each timestamp is an interval, [*begin time*, *end time*]. An annotation has two parts, a begin time constructor and an end time constructor. So an annotation has the following general form.

```
@temporal [begin time constructor; end time constructor]
```

The `@temporal` annotation specifies that this is a *temporal* semantics, rather than some other kind of metadata, such as *privacy*. Each constructor is an expression composed of temporal constructors (such as the functions, `min`, `max`, and `intersects`), time literals, and timestamp references. A timestamp reference has two potential forms. First, a reference can precisely name a timestamp by its position in the list of timestamps. As an example, `@1` refers to the first timestamp in the list, `@1.b` to the begin time of the first timestamp, and `@1.e` to the end time. As an example the constructor,

```
max(@1.b, @2.e)
```

computes the maximum of the begin time of the first timestamp and the end time of the second timestamp in the list of timestamps.

More common however, is a semantics applied to lists of indefinite length. The second form is used to iterate over the list with `@c` used to refer to the current timestamp, and `@p` used to refer to the previously computed (in the iteration) timestamp. As an example, the constructor

```
min(@p.b, @c.b)
```

computes the minimum time of the current begin time with the previously computed begin time. The reference `@c` is assumed by default so an equivalent formulation is

```
min(@p.b, @b)
```

where `@b` is the same as `@c.b`.

When iteration is used, the constructor is split into two parts, a base case, and an iterative case, i.e., each timestamp constructor has the following form: *base case*; *recursive case*. As an example,

```
@b ; min(@p.b, @b)
```

will compute the minimum of the begin times of the list of timestamps (recursive case), starting with the begin time of the first timestamp (base case).

The list of timestamps usually comes from the cartesian product of tables in the FROM clause of an SQL query. Each table has (or is given) a table variable. So the list of timestamps can be referred to by the corresponding list of table variables. As an example, consider the following FROM clause.

```
FROM Workers, Employees E
```

Table variables can be automatically generated for `Workers` resulting in the following FROM clause.

```
FROM Workers A0, Employees E
```

and the resulting list of table variables

```
(A0, E)
```

If the semantics

```
min(@1.b, @2.e)
```

is used for this list, the table variables can be substituted for the references to build an expression using actual columns.

```
min(A0.b, E.e)
```

4.2 Formal Specificatio of the Annotation Language

We propose specifying the semantics in a query language, like SQL, using annotations. The annotations have the following components.

- `@temporal` *specification* The semantics annotation specifies the kind of semantics. The semantics is described by the *specification*.
- *specification* - The specification is a constructor for the start and end times of a timestamp. It has one of the following forms:

  [*start*;*end*]

  or

  [*startBaseCase*;*startRecusiveCase*;*endBaseCase*;*endRecursiveCase*].

  The first form is the simple case where *start* and *end* are functions that each construct a time. The second form is for a list of timestamps, *startBaseCase*, *startRecursiveCase*, *endBaseCase*, and *endRecursiveCase* are functions that construct times. The semantics specifies a *fold* higher-order function (also commonly known as *reduce*, *foldl*, or *foldr*) for an arbitrary list of timestamps. The time construction functions are built from the following components.

  - @*t*.b - Refers to a tuple's beginning timestamp value.
  - @*t*.e - Refers to a tuple's ending timestamp value.
  - `@c.b` - Refers to the start time for the current timestamp in a list.
  - `@b` - Same as `@c.b`.

- `@c.e` - Refers to the end time for the current timestamp in a list.
- `@e` - Same as `@c.e`.
- `@p.b` - Refers to the previously folded timestamp's begin time.
- `@p.e` - Refers to the previously folded timestamp's end time.
- `@`n`.b` - Refers to the begin time of the nth timestamp.
- `@`n`.e` - Refers to the end time of the nth timestamp.
- `MIN_TIME` - Represents the minimum time.
- `MAX_TIME` - Represents the maximum time.
- `NOT_VIABLE` - Represents a non-viable time.
- `=,<,<=,+,−,max,min`, etc. - Assume integer operators, *e.g.,* as available in Java.
- (c) `?` e_1 `:` e_2 - Conditional operator, if condition c is true, produces e_1, else e_2.

We could extend the language with operators from Allen's algebra [26], or other temporal constructors, but this simple specification is suitable for our purposes.

Below we give specifications for some of the semantics discussed in this thesis. Each of the following is a *fold*-style annotation.

- Sequenced - `[MIN_TIME; max(@p.b,@b); MAX_TIME; min(@p.e,@e)]`
- Nonsequenced - `[MIN_TIME; MIN_TIME; MAX_TIME; MAX_TIME]`
- Preceding - `[MIN_TIME; max(@p.b,@b); MAX_TIME; MAX_TIME]`
- Context 3-7, sequenced - `[3; max(@p.b,@b); 7; min(@p.e,@e)]`

An annotation always specifies how to construct a time. But some constructed times are not viable, that is, when the begin time is after the end time, then the constructed time is not viable. The final semantics (Context 3-7, sequenced) may produce nonviable timestamps when the times do not fall within the range 3-7.

4.2.1 Annotation Examples

Let's consider some examples, to show how a `@temporal` annotation works. The following query projects Shoe employees using sequenced semantics.

```
@temporal [MIN_TIME; max(@p.b,@b); MAX_TIME; min(@p.e,@e)]
SELECT dept
FROM employees
WHERE salary >= 40000
```

The annotation is effectively a temporal statement modifier [4]. The annotation specifies the constructed timestamp for each group of lineage tuples, which is the timestamp of the result tuple. Projection is straightforward because there is only one group for each result tuple (recall that this is projection without duplication elimination). Only two tuples qualify for the result as shown in Figure 3.2(a). The timestamp for lineage tuple **id**(a) is `[1,7]`. We fold sequenced semantics using the previously folded timestamp (the base case is is `[MIN_TIME, MAX_TIME]`). So for the first timestamp the fold function computes

```
[max(MIN_TIME,1), min(MAX_TIME,7)]
```

which yields `[1,7]` for the timestamp of the result. The same logic is applied to the second result tuple.

Now consider the join result in Figure 3.2(c). The first tuple in the result has the lineage `(a,f)`. The semantics is applied to each timestamp in the lineage combination (in order). So first `a` is folded.

```
[max(MIN_TIME,1), min(MAX_TIME,7)]
```

Next `b` is folded.

```
[max(1,3), min(7,5)]
```

Yielding a timestamp of `[3,5]` as shown in Figure 3.2(c).

Finally let us consider the second query, to find the maximum salary. This query involves groups of lineage. Recall that the fold function computes a timestamp for the group, but lineage

must be grouped first, and the timestamp plays a role in the grouping. Group timestamps are always computed by intersection since they are involved in merging. Let us consider how the maximum salary is computed. First tuple a is chosen and placed in a group. Sequenced semantics stipulates that the group's timestamp becomes [1,7]. Next tuple b is chosen. Tuple b's timestamp does not intersect with the group $\{\texttt{a}\}^{[1,7]}$ so we add a new group yielding.

$$\{\{\texttt{a}\}^{[1,7]}, \{\texttt{b}\}^{[8,9]}\}$$

After tuple c is added we have.

$$\{\{\texttt{a}\}^{[1,5]}, \{\texttt{a}, \texttt{c}\}^{[6,7]}, \{\texttt{b}, \texttt{c}\}^{[8,9]}\}$$

Finally, after tuples d and e are grouped, we get.

$$\{\{\texttt{a}, \texttt{d}\}^{[1,2]}, \{\texttt{a}\}^{[3,5]}, \{\texttt{a}, \texttt{c}\}^{[6,7]}, \{\texttt{b}, \texttt{c}, \texttt{e}\}^{[8,9]}\}$$

Note that the timestamp for each group is affixed to the aggregate result.

In summary, lineage and timestamps interact in two ways. First, the timestamp for a tuple is computed, or combination of tuples, is computed using the semantics. Next, groups are determined using intersection semantics.

As a second example, consider the preceding semantics grouping for the maximum salary computation. First tuple a is chosen and placed in a group. Preceding semantics stipulates that the group's timestamp becomes [1,MAX_TIME]. Next tuple b is chosen yielding.

$$\{\{\texttt{a}\}^{[1,7]}, \{\texttt{a}, \texttt{b}\}^{[8,\texttt{MAX_TIME}]}\}$$

After tuple c is added we have.

$$\{\{\texttt{a}\}^{[1,5]}, \{\texttt{a}, \texttt{c}\}^{[6,7]}, \{\texttt{a}, \texttt{b}, \texttt{c}\}^{[8,\texttt{MAX_TIME}]}\}$$

Finally, after tuples d and e are grouped, we get.

$$\{\{\mathtt{a},\mathtt{d}\}^{[1,5]},\{\mathtt{a},\mathtt{c},\mathtt{d}\}^{[6,7]},\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\}^{[8,\mathtt{MAX_TIME}]}\}$$

The computation of the timestamp as specified by the semantics influences the grouping.

4.3 Precise Manipulation of Timestamps

The viable/nonviable distinction allows us to precisely control a semantics. *Precise* semantics manipulate a fixed number of timestamps (*i.e.,* supports *explicit* nonsequenced semantics). We now give some specifications for some precise semantics. Recall that `@b.1` refers to the begin time of the first timestamp, `@e.1` refers to the end time of the first timestamp, while `@b.2` refers to the begin time of the second timestamp, and so on.

- Timestamp 1 meets timestamp 2

```
[(@2.b == @1.e) ?  @1.b : NOT_VIABLE; @2.e]
```

- Timestamp 1 is before timestamp 2

```
[(@2.b < @1.e) ?  @1.b : NOT_VIABLE; @2.e]
```

For example, suppose that we want to find Shoe employees who got a raise and when they got that raise (a Shoe employee tuple meets another with a higher salary).

```
@temporal [(@2.b == @1.e) ?  @1.b : NOT_VIABLE; @2.e]
SELECT A.name
FROM employees A, employees B
WHERE A.dept = 'Shoes'
  AND B.dept = 'Shoes'
  AND A.name = B.name
  AND A.salary < B.salary
```

The query manipulates the timestamps of the combination of tuples in the FROM clause (as specified by the annotation). Note that a semantics must always construct a timestamp, so the result is always a temporal relation (unlike in traditional nonsequenced semantics [27]). Since the semantics has access to the timestamps and can enforce a precise semantics on the query, we do not see a need for explicit functions embedded in a query. The query is imbued as *temporal* by annotating it.

4.3.1 Sequenced Support is Key

We close this section by observing that lineage offers a way to implement the temporal semantics. Dignös et al. showed how to engineer Postgres to support sequenced semantics [18]. For a subset of SQL, which we style SQL--, which consists of the constructive parts of SQL consisting of projection, join, selection, grouping, and aggregation, implementation is relatively straightforward. For the eliminative parts of SQL, such as some subqueries, difference, and outer join, changes are needed in the DBMS to support sequenced semantics for tuples timestamped with intervals. But the same kinds of changes are needed to support *any sequenced semantics*. That is, if sequenced semantics is supported, then all of the other semantics should be able to be supported using the same techniques.

We consider a new approach in the next chapter.

CHAPTER 5

TRANSLATION

In this chapter we describe how temporal SQL is translated to nested SQL, which is described in Chapter 2.3. We focus only on the SELECT statement in SQL, which is how SQL users query data. We first give an example to illustrate the translation process, then a denotational semantics for the SELECT statement translation is provided.

5.1 Translation Example

In Temporal SQL an SQL query can be annotated with a temporal semantics. The annotation converts the query to a temporal query since the query is evaluated using the temporal semantics specified in the annotation. As our first example, we translate a simple SELECT statement that computes the Cartesian product of two relations.

```
@temporal [MAX_TIME; max(@b,@p.b); MIN_TIME; min(@e,@p.e)]
SELECT *
FROM r,s;
```

In the above example we are adding the annotation for sequenced semantics to a simple `SELECT` statement. The translated query is given below.

```
SELECT *, max(A1.begin,max(MAX_TIME,A0.begin)) as begin,
          min(A1.end,min(MIN_TIME,A0.begin)) as end
FROM r A0, s A1
WHERE max(A1.begin,max(MAX_TIME,A0.begin)) >=
                   min(A1.end,min(MIN_TIME,A0.begin))
```

The translation adds table variables `A0` and `A1` to the `FROM` clause so that fields in each tuple from a table can be referenced elsewhere in the query. The translation also adds a predicate (the `WHERE` clause) to determine if a viable timestamp can be computed by the pair of tuples chosen to be in

the Cartesian product. If the tuple from relation `r` does not *intersect* with the tuple from relation `s` then the two tuples were never in the database as the same time and hence do not produce a result tuple. The sequenced semantics annotation specifies that temporal intersection should be computed. Finally, to the `SELECT` clause is added constructors for the `begin` and `end` times in the result.

5.2 A Denotational Semantics for Temporal SQL

Our goal is to translate temporal SQL to nested SQL. The formal semantics of the translation is given as a denotational semantics. A denotational semantics is an approach of formalizing the meanings of programming languages constructs by describing their meanings as mathematical objects (called denotations).

Let's consider the denotational semantics for a basic SELECT statement.

$$
\begin{array}{l}
[\![\texttt{@temporal}\ S \\
\ \texttt{SELECT}\ A_1, \ldots, A_n \\
\ \texttt{FROM}\ R_1, \ldots, R_n \\
\ [\texttt{WHERE}\ P] \\
]\!] \equiv \\
\quad \texttt{SELECT}\ A_1, \ldots, A_n, \mathtt{T_A}(\mathtt{S}, [\alpha_1, \ldots, \alpha_\mathtt{n}]) \\
\quad \texttt{FROM}\ \alpha_1 = [\![R_1]\!], \ldots, \alpha_1 = [\![R_n]\!] \\
\quad [\texttt{WHERE}\ P\ \texttt{AND}\ \mathtt{T_P}(S, [\alpha_1, \ldots, \alpha_n])]
\end{array}
$$

$$
\begin{array}{l}
[\![R]\!] \equiv R\ L()\ \text{returns}\ \alpha \\
[\![R\ L]\!] \equiv R\ L\ \text{returns}\ L
\end{array}
$$

In the translation, a temporal attribute is added to the `SELECT` clause

$$\texttt{SELECT}\ A_1, ..., A_n, T_A(S, [\alpha_1, \ldots, \alpha_n])$$

$T_A(S, [\alpha_1, \ldots, \alpha_n])$ is a semantic function, called the *temporal attribute function*, that takes a temporal semantics annotation, S, and a list of table references, $[\alpha_1, \ldots, \alpha_n]$ and constructs a pair of

times. A temporal predicate is also added to the `WHERE` clause (or if the `WHERE` clause is absent, it is created) as follows.

`WHERE` P `AND` $T_P(S, [\alpha_1, ..., \alpha_n])$

$T_P(S, [\alpha_1, ..., \alpha_n])$ is also a semantic function, called the *temporal predicate function*, that is added to the `WHERE` clause. The temporal predicate function ensures that the begin time is before or equal to the end time, that is, that the constructed interval is a viable interval.

For example, consider the following translation.

```
⟦@temporal[max(@1.b,@2.b); min(@1.e,@2.e)]
  SELECT B.X, C.Y
  FROM B, C
  WHERE B.X > C.Y⟧
⟧ ≡
  SELECT B.X, C.Y, T_A([max(@1.b,@2.b); min(@1.e,@2.e)],[α1,α2])
  FROM α1 = ⟦B⟧, α2 = ⟦C⟧
  WHERE B.X > C.Y AND T_P([max(@1.b,@2.b); min(@1.e,@2.e)],[α1,α2])
⟧ ≡
  SELECT B.X, C.Y, T_A([max(@1.b,@2.b); min(@1.e,@2.e)],[A1,A2])
  FROM B A1, C A2
  WHERE B.X > C.Y AND T_P([max(@1.b,@2.b); min(@1.e,@2.e)],[A1,A2])
⟧ ≡
  SELECT B.X, C.Y,
         max(A1.begin, A2.begin) as begin,
         min(A1.end, A2.end) as end
  FROM B A1, C A2
  WHERE B.X > C.Y AND
         max(A1.begin, A2.begin) <= min(A1.end, A2.end)
```

The temporal attribute and predicate functions utilize the temporal semantics to specify begin and end time constructors. T_P tests whether the constructed begin time is before or equal to the constructed end time, while T_A produces a pair of attribute values: the begin time and the end time values.

5.2.1 INTERSECT

The SQL INTERSECT clause is used to evaluate the intersection of two SELECT statements. It returns each row from the evaluation of the first SELECT statement that is identical to some row in the evaluation of the second SELECT statement.

$[\![$@temporal S
 (SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ [WHERE P_R])
 INTERSECT
 (SELECT $A_1, \ldots, A_n$ FROM $S_1, \ldots, S_n$ [WHERE P_S])
$]\!] \equiv$
  SELECT $A_1, \ldots, A_n$, TIME
  FROM FLATTEN (
   SELECT $A_1, \ldots, A_n$, TS_INTERSECTION(α_1.TIME,α_2.TIME) AS TIME
   FROM
    ($[\![$@temporal S SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ [WHERE P_R]$]\!]$) $\alpha_1 = L(\,)$
    COGROUP
    ($[\![$@temporal S SELECT $A_1, \ldots, A_n$ FROM $S_1, \ldots, S_n$ [WHERE P_S]$]\!]$) $\alpha_2 = L(\,)$
    ON $(A_1, \ldots, A_n)$
) WHERE TIME IS NOT NULL

5.2.2 EXCEPT

The SQL EXCEPT clause/operator performs set difference between two relations, that is, EXCEPT returns only rows in the first SELECT, which are not in second SELECT statement.

$[\![$@temporal S

(SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ [WHERE P_R])

EXCEPT

(SELECT $A_1, \ldots, A_n$ FROM $S_1, \ldots, S_n$ [WHERE P_S])

$]\!] \equiv$

SELECT $A_1, \ldots, A_n$,TIME

FROM FLATTEN (

SELECT $A_1, \ldots, A_n$,TS_DIFFERENCE(α.TIME)

FROM

([[@temporal S SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ [WHERE P_R]]) $\alpha_1 = L(\,)$

COGROUP

([[@temporal S SELECT $A_1, \ldots, A_n$ FROM $S_1, \ldots, S_n$ [WHERE P_S]]) $\alpha_2 = L(\,)$

ON $(A_1, \ldots, A_n)$

) WHERE TIME IS NOT NULL

5.2.3 UNION

The SQL UNION clause is used to perform the union of two SELECT statements.

[[@temporal S

(SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ [WHERE P_R])

UNION

(SELECT $A_1, \ldots, A_n$ FROM $S_1, \ldots, S_n$ [WHERE P_S])

$]\!] \equiv$

SELECT $A_1, \ldots, A_n$, TIME

FROM FLATTEN (

SELECT $A_1, \ldots, A_n$, TS_COALESCE(α.TIME)

FROM

([[@temporal S SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_n$ [WHERE P_R]]) $L(\,)$

UNION

([[@temporal S SELECT $A_1, \ldots, A_n$ FROM $S_1, \ldots, S_n$ [WHERE P_S]]) $L(\,)$

```
    GROUP BY(A_1, ..., A_n) α = L( ))
)
```

5.2.4 Subqueries

A subquery is a query in the WHERE clause of a SELECT statement. There are three kinds of subqueries based on the kind of result they produce: scalar, single column, or multi-column. Different keywords are used to compare an outer query value with the result produced by a subquery. A scalar producing subquery produces a single value, so scalar comparison predicates, *e.g.,* equals, can be used. A single-column producing subquery essentially produces a list so membership `IN` the list can be tested, or other comparisons like a value is `> ALL` the values in the list. A multicolumn subquery can only be tested to determine whether it produces a result, using the `EXISTS` keyword. A subquery might also be *correlated*.

A *correlated* subquery references a table variable from the outer query. Correlated subqueries are the same as non-correlated queries for our purposes, assuming the inner and outer queries share the same annotation.

For our purposes, the interesting part about a subquery is that the result it produces varies over time. So a scalar producing subquery produces a single value at any time t, and at different times could produce different values.

Single-column producing Subqueries

Single-column producing subqueries produce a list of values. Different operators are used to test the list.

The outer query can test to see if an expression, X, is `IN` the result of a single-column producing subquery.

```
[[@temporal S
 SELECT A_1, ..., A_n
 FROM R_1, ..., R_m
 WHERE C AND X IN (SELECT B FROM T)
```

]] ≡

[[@temporal S

(SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C)

INTERSECT

(SELECT T.A_1,...,T.A_n

FROM (SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C) A0 JOIN T ON (X))]]

The intuition is that the subquery behaves like a (left) semi-join. The second operand in the INTERSECT performs a superset of the semi-join, since duplicates could be created. To remove the duplicates obeying the temporal semantics, we perform an INTERSECT with the original relation.

The NOT IN operator used with a subquery translates to an EXCEPT rather than an INTERSECT operation as follows.

[[@temporal S

SELECT $A_1, \ldots, A_n$

FROM $R_1, \ldots, R_m$

WHERE C AND X IN (SELECT B FROM T)

]] ≡

[[@temporal S

(SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C)

EXCEPT

(SELECT T.A_1,...,T.A_n

FROM (SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C) A0 JOIN T ON (X))]]

The intuition is to remove from the outer query all of the times that X is in the subquery (produced by the semi-join).

Another form uses a comparison to a value, *e.g.,* <= ALL. We refer to this as P ALL below.

[[@temporal S

SELECT $A_1, \ldots, A_n$

FROM $R_1, \ldots, R_m$

WHERE C AND $X\ P$ (SELECT B FROM T)

$\rrbracket \equiv$

$\llbracket$@temporal S

(SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C)

EXCEPT

(SELECT T.$A_1, \ldots,$T.A_n

FROM (SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C) A0 JOIN T ON $(\not(X\ P\ B)))\rrbracket$

The intuition is that we compute values and times when the negation of the comparison P holds. We remove these values and times from the outer query using EXCEPT.

The P ANY version of the subquery is similar, but uses INTERSECT. We omit the denotation for brevity.

Scalar-producing subquery

A scalar producing subquery produces a result that is then compared to a value in the outer query using comparison operator P, *e.g.,* >=.

$\llbracket$@temporal S

SELECT $A_1, \ldots, A_n$

FROM $R_1, \ldots, R_m$

WHERE C AND $X\ P$ (SELECT B FROM T)

$\rrbracket \equiv$

$\llbracket$@temporal S

(SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C)

INTERSECT

(SELECT T.$A_1, \ldots,$T.A_n

FROM (SELECT $A_1, \ldots, A_n$ FROM $R_1, \ldots, R_m$ WHERE C) A0 JOIN T ON $(X\ P\ B))\rrbracket$

The second operand in the INTERSECT produces tuples and times where $X\ P\ v$ (v is the value produced by the subquery) holds. For those times, the INTERSECT chooses the appropriate tuples from the outer query.

Multi-column producing subquery

A multi-column subquery can be tested to determine if it produces any tuples (using `EXISTS`) or produces nothing (using `NOT EXISTS`). The subquery may produce tuples at some times, but not at others. The intuition to the translation is to select tuples in the outer relation that live at the same time as some tuple in the inner relation. Note that because of correlated subqueries the inner relation is computed for every tuple in the outer relation.

⟦`@temporal` S
`SELECT` $A_1, \ldots, A_n$
`FROM` $R_1, \ldots, R_m$
`WHERE` C `AND EXISTS` (`SELECT` $B_1, \ldots, B_i$ `FROM` $T_1, \ldots, T_j$ `WHERE` F)
⟧ ≡
⟦`@temporal` S
`(SELECT` $A_1, \ldots, A_n$ `FROM` $R_1, \ldots, R_m$ `WHERE` C`)`
`INTERSECT`
`(SELECT T.`A_1`,...,T.`A_n
`FROM (SELECT` $A_1, \ldots, A_n$ `FROM` $R_1, \ldots, R_m, B_1, \ldots, B_i$ `WHERE` C `AND` F `) T )`⟧

The `NOT EXISTS` form replaces `INTERSECT` with `DIFFERENCE`.

5.2.5 Outer Join

A left outer join returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link field. Right outer join is the symmetric case and full outer join is the union of the left and right outer joins. We give the left outer join translation below.

⟦`@temporal` S
`SELECT` $A_1, \ldots, A_n$
`FROM` R `LEFT OUTER JOIN` S `ON (J)`
[`WHERE` P]
⟧ ≡

```
⟦@temporal S
SELECT A_1, ..., A_n
FROM R JOIN S ON (J)
[WHERE P] ⟧
UNION
SELECT A_1, ..., A_n,TIME FROM
(SELECT A_1, ..., A_n
FROM ( R
    EXCEPT
    (SELECT R.*
      FROM R JOIN S ON(J)
      [WHERE P]) Q⟧
) R LEFT OUTER JOIN S
```

5.2.6 Grouping and Aggregation

This area has been well-studied in previous research in temporal databases [28]. We leave grouping and aggregation for future work.

5.3 Examples

In this section we present some examples of the translation, and how it works in the evaluation of a query.

Now let's focus on the translation and evaluation of a temporal INTERSECT. Consider the relations in Figure 5.1. First, let's evaluate a sequenced temporal INTERSECTION. As the first step, COGROUP is applied to the relations, yielding the relations shown in Figure 5.2. Next, the relations are COALASCEd, resulting in the intermediate result of Figure 5.3 Then the coalesced relations are INTERSECTed (Figure 5.4. Finally, the relations are FLATTENed (Figure 5.5).

Now let's consider a preceding EXCEPT. Initially, the timestamps are interpreted as given in Figure 5.6. Again, the first step is to COGROUP as shown in Figure 5.7. Then the relations are

Employee Table One

dept	time
Shoes	[1,7]
Hats	[8,9]
Shoes	[6,9]
Shoes	[1,2]
Camera	[8,9]

Employee Table Two

dept	time
Hats	[1,14]
Shoes	[1,6]
Shoes	[8,11]
Camera	[3,14]
Tape	[2,5]

Fig. 5.1: Example Tables of Employees

dept	time
Shoes	{(Shoes,[1,7]),(Shoes,[6,9]), (Shoes,[1,2])}
	{(Shoes,[1,6]),(Shoes,[8,11])}
Hats	{(Hats,[8,9])},{(Hats,[1,14])}
Camera	{(Camera,[8,9])},{(Camera,[3,14])}
Tape	{ (Tape,[2,5])}

Fig. 5.2: After COGROUP

dept	time
Shoes	{ (Shoes,[1,9])},
	{(Shoes,[1,6]),(Shoes,[8,11])}
Hats	{(Hats,[8,9])}, {(Hats,[1,14])}
Camera	{ (Camera,[8,9])},{(Camera,[3,14])}
Tape	{(Tape,[2,5])}

Fig. 5.3: After COALESCE

dept	time
Shoes	(Shoes,[1,6],(Shoes,[8,9])
Hats	(Hats,[8,9])
Camera	(Camera,[8,9])
Tape	(Tape,[2,5])

Fig. 5.4: After INTERSECT

dept	time
Shoes	[1,6]
Shoes	[8,9]
Hats	[8,9]
Camera	[8,9]
Tape	[2,5]

Fig. 5.5: After FLATTEN

COALESCED (Figure 5.8) and the EXCEPT is done (Figure 5.9).

dept	time
Shoes	[1,MAX]
Hats	[8,MAX]
Shoes	[6,MAX]
Shoes	[1,MAX]
Camera	[8,MAX]

dept	time
Hats	[1,MAX]
Shoes	[1,MAX]
Shoes	[8,MAX]
Camera	[3,MAX]
Tape	[2,MAX]

Fig. 5.6: Preceding Except Tables

dept	time
Shoes	{[1,MAX],[6,MAX]}, {[1,MAX],[8,MAX]}
Hats	{[8,MAX]},{[1,MAX]}
Camera	{[8,MAX]},{[3,MAX]}
Tape	{[2,MAX]}

Fig. 5.7: Preceding COGROUP

Finally, let's consider a nonsequenced UNION. Initially, the timestamps are interpreted as given in Figure 5.10. Again, the first step is to COGROUP as shown in Figure 5.11. Finally, the relations are FLATTENed yielding the result shown in Figure 5.12.

dept	time
Shoes	{[1,MAX]},{[1,MAX]}
Hats	{[8,MAX]},{[1,MAX]}
Camera	{[8,MAX]},{[3,MAX]}
Tape	{[2,MAX]}

Fig. 5.8: Preceding COALESCE

dept	time
Hats	[1,7]
Camera	[3,7]
Tape	[2,MAX]

Fig. 5.9: Preceding EXCEPT Result

dept	time
Shoes	[MIN,MAX]
Hats	[MIN,MAX]
Shoes	[MIN,MAX]
Shoes	[MIN,MAX]
Camera	[MIN,MAX]

dept	time
Hats	[MIN,MAX]
Shoes	[MIN,MAX]
Shoes	[MIN,MAX]
Camera	[MIN,MAX]
Tape	[MIN,MAX]

Fig. 5.10: Nonsequenced Interpretation

dept	time
Shoes	[MIN,MAX],[MIN,MAX]
Hats	[MIN,MAX],[MIN,MAX]
Camera	[MIN,MAX],[MIN,MAX]
Tape	[MIN,MAX]

Fig. 5.11: Nonsequenced COGROUP

dept	time
Shoes	[MIN,MAX]
Hats	[MIN,MAX]
Camera	[MIN,MAX]
Tape	[MIN,MAX]

Fig. 5.12: Nonsequenced FLATTEN

CHAPTER 6
IMPLEMENTATION

Our Goal is to translate Temporal SQL to Nested SQL. The translation is a two step process. First, what should go into the translation. Second, where should the translated code go.

To understand the above two steps let us consider a temporal SELECT statement using denotational semantics.

```
@temporal [max(1,(max(@1.b,@2.b)); min(7,min(@1.e,@2.e))]
SELECT * from r,s;
```

The translated SELECT is given below.

```
SELECT *, max(1,(max(A0.begin,A1.begin)) AS begin,
          min(7,min(A0.end,A1.end)) AS end
FROM r A0,s A1
WHERE max(1,(max(A0.begin,A1.begin)) <= min(7,min(A0.end,A1.end))
```

The first step is to generate labels for the tables listed in the FROM clause.

```
FROM r A0,s A1
```

These labels are used in the other parts of the translation. Next, the temporal attribute function, T_A, and temporal predicate function, T_P, are computed. T_A evaluates to the following, which is appended to the SELECT clause to project the “hidden” temporal attributes for the result.

```
max(1,(max(A0.begin,A1.begin)) AS begin,
min(7,min(A0.end,A1.end)) AS end
```

T_P generated code to ensure that the start time is less than or equal to the end time. It evaluates to the following, which is added as a conjunct in the WHERE clause.

```
max(1,(max(A0.begin,A1.begin)) <= min(7,min(A0.end,A1.end))
```

6.1 Translation using SQLite and ANTLR

To implement the translation we choose to use the SQLite grammar. SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite has an ANTLR, version two, grammar. ANTLR is a parser generation tool, similar to a combination of LEX and YACC. ANTLR is pure Java and generates a parser in Java.

To implement the translation we rewrote the token stream. ANTLR first converts a program (text) into a stream of Token objects. It is easy to manipulate the program by inserting into, deleting from, and modifying the tokens using the following methods.

- InsertBefore(Token t, Object text): InsertBefore tokenstream goes to the token 't' and adds 'text' to it before the end of token.
- InsertAfter(Token t, Object text): InsertAfter tokenstream goes to the token 't' and adds 'text' to it after the token.
- Replace(Token t, Object text): Replace tokenstream replaces the token 't' value with the 'text' that is being inserted.
- itemize

The token stream rewrites were woven into the grammar as semantic actions. Consider the grammar rule for aliasing the table names.

```
table_or_subquery :
  ( database_name '.' )? table_name
  ( K_AS? table_alias )?
```

We need to add semantic actions to generate a new label or capture the existing table variable and add it to the list of table variables. This is complicated by the potential for name clashes. A name clash occurs when a generated table variable is used elsewhere in a query. First we have to capture the table name token (for other purposes).

```
table_or_subquery :
  ( d=database_name '.' )? t=table_name
```

The above line does the job for us. `t` as the table name token. We add the text of the token to the list of table names. In case there is no alias name we will assign a unique name to it. The below code performs that task.

```
{
  // Capture the table name
  tableNameList.add((($d)? $d.text . "." : "") . $t.text);
}
```

In case we have an alias name we need to grab its token, as follows.

```
K_AS? r=table_alias
```

`r` as the table alias. If the alias exists, then we should use the alias, otherwise we should generate a new alias, or rather remember in the token stream where a new alias should be placed.

```
{
  if ($r != null) {
    // have alias, add to list
    aliasNames.put(new Alias($r.text));
    // maintain a set of previously used table names
    aliasNameSet.add($r.text);
  } else {
    // generate a previously unused alias, update alias set
    Alias alias = generateAlias(aliasNameSet,alias);
    // use the alias
    aliasNames.put(alias);
    // insert into list of token stream places to update
    aliasTokenList.add(new Pair(t,alias));
  }
}
```

Putting it all together we get the following rule.

```
table_or_subquery :
  ( d=database_name '.' )? t=table_name
  {
    // Capture the table name
    tableNameList.add((($d)? $d.text . "." : "") . $t.text);
  }
K_AS? r=table_alias
  {
    if ($r != null) {
      // have alias, add to list
      aliasNames.put(new Alias($r.text));
      // maintain a set of previously used table names
      aliasNameSet.add($r.text);
    } else {
      // generate a previously unused alias, update alias set
      Alias alias = generateAlias(aliasNameSet,alias);
      // use the alias
      aliasNames.put(alias);
      // insert into list of token stream places to update
      aliasTokenList.add(new Pair(t,alias));
    }
  }
```

As an example, suppose we are considering the below temporal SELECT statement.

```
@temporal [max(1,(max(@1.b,@2.b)); min(7,min(@1.e,@2.e))]
SELECT *
FROM r, s, t A1;
```

Then we will get the following output based on the above grammar. `A0` and `A2` are generated table aliases. `A1` would have been generated but already existed as a table alias so `A2` was generated

instead.

```
SELECT *
FROM r A0, s A2, t A1;
```

CHAPTER 7
CONCLUSION AND FUTURE WORK

Temporal databases are databases that have special capabilities for handling time. Prior research identified two common semantics for temporal operations: sequenced and nonsequenced. Sequenced semantics evaluates an operation in each time instant using only the data alive at that time. Nonsequenced semantics, in contrast, means that an operation explicitly references and manipulates the timestamps in the data. The semantics were considered as very different.

In this thesis we proposed a novel framework that shows both sequenced semantics and nonsequenced semantics are variants of a general *temporal semantics*. The general semantics uses different *slices* of the database. A slice is the data “alive” at a given time. At its core sequenced semantics considers the data alive in an instantaneous *slice*, that is only data whose lifetime overlaps the instant is part of the database state at that instant. Nonsequenced semantics then has a different slice. All of the data in the database is alive at every instant. Hence, they are both variants of a general temporal semantics. Other semantics, such as preceding semantics can be defined using different slices. Preceding semantics is the data alive up to a given time.

The primary contribution of this thesis is the translation from *temporal SQL* to *nested SQL*. In order to do the translation we use annotations. In Temporal SQL an SQL query can be annotated with a temporal semantics. The annotation imbues the query with a temporal interpretation. The query is evaluated using the temporal semantics specified in the annotation. We adopted source-to-source translation layer, and translated the output to nested SQL rather than SQL. “Nested SQL” is an SQL with non-1NF constructs, necessary for computing with bags of tuples and timestamps.

We provide a *denotational semantics* for translating *temporal SQL* to *nested SQL*. We implemented the denotational semantics using an SQLite ANTLR grammar. We gave a denotational semantics for SQL-92, except for grouping and aggregation. Temporal grouping and aggregation have been previously researched [28].

In conclusion, we showed that it was possible to translate temporal SQL to nested SQL. But

much remains to be done. In particular, the key remaining challenge is to implement nested SQL. We think that our strategy for implemented temporal SQL can be reused for other kinds of metadata, e.g., privacy, security, lineage, etc. So the blueprint we developed could be used for translating lineage SQL to nested SQL. But supporting the plethora of metadata requires an implementation of nested SQL.

We also need to consider post SQL-92 extensions of SQL, such as CUBE BY and windowing functions. The SQL language is a moving target and we have only considered the basic parts of SQL's query construct.

REFERENCES

[1] M. H. Böhlen and C. S. Jensen, "Sequenced semantics," in *Encyclopedia of Database Systems*, 2009, pp. 2619–2621. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_1053

[2] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Nonsequenced semantics," in *Encyclopedia of Database Systems*, 2009, pp. 1913–1915. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_1052

[3] R. T. Snodgrass, "The temporal query language tquel," *ACM Trans. Database Syst.*, vol. 12, no. 2, pp. 247–298, 1987. [Online]. Available: http://doi.acm.org/10.1145/22952.22956

[4] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Temporal Statement Modifiers," *ACM Trans. Database Syst.*, vol. 25, no. 4, pp. 407–456, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=377674.377665

[5] C. S. Jensen and R. T. Snodgrass, "Snapshot equivalence," in *Encyclopedia of Database Systems*, 2009, p. 2659. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_1417

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

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

[8] C. E. Dyreson, "Observing transaction-time semantics with ttxpath," in *WISE (1)*, 2001, pp. 193–202. [Online]. Available: http://dx.doi.org/10.1109/WISE.2001.996480

[9] F. Grandi, "T-SPARQL: A tsql2-like temporal query language for RDF," in *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information*

Systems, Novi Sad, Serbia, September 20-24, 2010, 2010, pp. 21–30. [Online]. Available: http://ceur-ws.org/Vol-639/021-grandi.pdf

[10] F. Rizzolo and A. A. Vaisman, "Temporal XML: modeling, indexing, and query processing," *VLDB J.*, vol. 17, no. 5, pp. 1179–1212, 2008. [Online]. Available: http://dx.doi.org/10.1007/s00778-007-0058-x

[11] C. X. Chen and C. Zaniolo, "Sqlst: A spatio-temporal data model and query language," in *ER*, 2000, pp. 96–111. [Online]. Available: http://dx.doi.org/10.1007/3-540-45393-8_8

[12] A. A. Vaisman and A. O. Mendelzon, "A temporal query language for OLAP: implementation and a case study," in *Database Programming Languages, 8th International Workshop, DBPL 2001, Frascati, Italy, September 8-10, 2001, Revised Papers*, 2001, pp. 78–96. [Online]. Available: http://dx.doi.org/10.1007/3-540-46093-4_5

[13] J. Chomicki and D. Toman, "Abstract versus concrete temporal query languages," in *Encyclopedia of Database Systems*, 2009, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_1559

[14] C. E. Dyreson, "Aspect-oriented relational algebra," in *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, 2011, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/1951365.1951411

[15] C. E. Dyreson, O. U. Florez, A. Thakre, and V. Sharma, "Supporting data aspects in pig latin," in *AOSD*, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2451436.2451439

[16] K. Torp, C. S. Jensen, and M. H. Böhlen, "Layered temporal DBMS: concepts and techniques," in *Database Systems for Advanced Applications '97, Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, April 1-4, 1997*, 1997, pp. 371–380.

[17] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Stratum approaches to temporal DBMS implementation," in *Proceedings of the 1998 International Database Engineering and*

Applications Symposium, IDEAS 1998, Cardiff, Wales, U.K., July 8-10, 1998, 1998, pp. 4–13. [Online]. Available: http://dx.doi.org/10.1109/IDEAS.1998.694346

[18] A. Dignös, M. H. Böhlen, and J. Gamper, "Temporal alignment," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, 2012, pp. 433–444. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213886

[19] 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*. Springer-Verlag, 1998, pp. 367–405.

[20] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the lineage of view data in a warehousing environment," *ACM Trans. Database Syst.*, vol. 25, no. 2, pp. 179–227, 2000. [Online]. Available: http://doi.acm.org/10.1145/357775.357777

[21] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," in *CIDR*, 2005, pp. 262–276. [Online]. Available: http://www.cidrdb.org/cidr2005/papers/P22.pdf

[22] "PIG Latin defenitions," https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html, accessed: 2016-02-20.

[23] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in temporal databases," in *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, 1996, pp. 180–191. [Online]. Available: http://www.vldb.org/conf/1996/P180.PDF

[24] C. E. Dyreson, "Temporal coalescing with now, granularity, and incomplete information," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, 2003, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/872757.872779

[25] M. H. Böhlen, "Temporal coalescing," in *Encyclopedia of Database Systems*, 2009, pp. 2932–2936. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_388

[26] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358434

[27] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner, "Transitioning temporal support in TSQL2 to SQL3," in *Temporal Databases, Dagstuhl*, 1997, pp. 150–194. [Online]. Available: http://dx.doi.org/10.1007/BFb0053702

[28] J. Gamper, M. H. Böhlen, and C. S. Jensen, "Temporal aggregation," in *Encyclopedia of Database Systems*, 2009, pp. 2924–2929. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_386

[29] D. Toman, "Point vs. interval-based query languages for temporal databases," in *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada*, 1996, pp. 58–67. [Online]. Available: http://doi.acm.org/10.1145/237661.237676

[30] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner, "Transitioning Temporal Support in TSQL2 to SQL3," in *Temporal Databases, Dagstuhl*, 1997, pp. 150–194. [Online]. Available: http://dx.doi.org/10.1007/BFb0053702

[31] Y. Cui and J. Widom, "Lineage tracing in a data warehousing system," in *ICDE*, 2000, pp. 683–684. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2000.839493

[32] K. G. Kulkarni and J. Michels, "Temporal features in SQL: 2011," *SIGMOD Record*, vol. 41, no. 3, pp. 34–43, 2012. [Online]. Available: http://doi.acm.org/10.1145/2380776.2380786

[33] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass, "On the Semantics of "Now" in Databases," *ACM Transactions on Database Systems*, vol. 22, no. 2, pp. 171–214, 1997.