Department of Informatics, University of Zürich

BSc Thesis

Query Compilation of Statement Modifiers

Oliver Leumann

Email: oliver.leumann@uzh.ch

September 14, 2013

supervised by Prof. Dr. M. Böhlen and A. Dignös

University of Zurich[UZH]

Department of Informatics

Contents

1 Introduction **5**

2 Background and Related Work **9**
2.1 General Preliminaries 9
2.2 Reduction Rules 9
2.3 The Alignment Operator 11
2.4 The Normalization Operator 13
2.5 Timestamp Propagation 14

3 Definition of an SQL Language with Statement Modifiers **15**
3.1 Basic Definition for Temporal Statement Modifiers 15
3.2 Scope of Statement Modifiers 17
3.3 Supporting Statement Modifiers for Set Operations 17
3.4 Additional Parameters for Statement Modifiers 21
3.5 Allowing the Scaling of Attribute Values 22

4 The SQL mapping **25**
4.1 Implementation of the Temporal Primitives 25
4.2 The Mapping of Queries with Statement Modifiers to Queries with Temporal Primitives over Algebraic Expressions 27
4.3 The Mapping of Queries with Statement Modifiers to Queries with Temporal Primitives over SQL Expressions 30
4.3.1 Preliminaries for the actual Mapping 30
4.3.2 Reducing Temporal Join-Expressions 33
4.3.3 Reducing Temporal Projection and Temporal Aggregation 34

5 Implementation **37**
5.1 Preliminaries 37
5.2 Determining the Scope of the Statement Modifier 38
5.3 The Reduction of Temporal Joins 39
5.3.1 Preventing Endless Reductions on Join-Expressions 39
5.3.2 Removing Duplicate Interval Timestamps 40
5.4 The Reduction of the Explicit Cartesian Product 43
5.5 The Reduction of Projection and Aggregation 44

6 Conclusion **46**

Abstract

For the querying of temporal relations statement modifiers have been proposed in the past. Statement modifiers can be prepended to an SQL query to indicate if the given query should be evaluated non-temporal or temporal by the database system. For the processing and evaluation of temporal database operators, such as aggregation and joins, reduction rules using temporal primitives have been proposed. The shortcoming of these approaches is that they are not yet combined. On one side a query language is defined that has not been implemented, and on the other side an evaluation mechanism for database operators based on relation algebra exists, that requires large and cumbersome SQL queries to express temporal statements. The focus of this thesis is to first provide the basis to smoothly combine the two approaches, i.e., to compile queries with statement modifiers to statements with temporal primitives, and then to implement the compilation mechanism into the database system PostgreSQL whose kernel has already been extended with the temporal primitives. For the compilation mechanism we have to ensure that the scaling of attribute values at query time is still possible.

Abstract

Für das Abfragen temporaler Relationen wurde in der Vergangenheit Statement Modifiers vorgestellt. Statement Modifiers können an eine SQL-Abfrage vorangestellt werden um anzugeben, ob diese Abfrage auf nicht-temporale oder temporale Weise vom Datenbanksystem ausgewertet werden soll. Für das Ausführen und Auswerten von temporalen Datenbankoperatoren wurden Reduktions Regeln eingeführt, welche auf temporale Primitiven zurückgreifen. Das Manko bei diesen Ansätzen ist, dass sie bis jetzt noch nicht kombiniert wurden. Das Hauptaugenmerk dieser Bachelorarbeit liegt auf der Kombination dieser beiden Ansätze, das heisst das Datenbankabfragen mit vorangestellten Statement Modifiers zu Abfragen mit temporalen Primitiven kompiliert werden sollen. Dabei soll der Kompilierungsmechanismus in das Datenbanksystem PostgreSQL implementiert werden, dessen Kernel bereits mit den temporalen Primitiven erweitert wurde.

1 Introduction

To make the support of temporal queries available, a lot of things have to be considered. Group based operators like projection, aggregation, union, difference and intersection as well as the tuple based operators selection, Cartesian product, inner join, left outer join, right outer join, full outer join and anti join need to be transformed appropriately. If interval timestamps of the argument relations are used in predicates and functions, this must be supported as well. Also some attribute values may not stay valid if the associated timestamps change and it is necessary that the scaling of such attributes are possible. Solutions to these topics have been proposed [1, 5] but to make all of them work with respect to temporal statement modifiers [2] new challenges result from such an attempt. The timestamp propagation and scaling of attributes need to be executed in the right place inside the order of reducing temporal operators. Also the determination of the timestamps as well as the scope of temporal statement modifiers have to be chosen wisely.

Example 1 *Consider the two non-temporal relations* $\mathbf{E}$ *and* $\mathbf{M}$ *shown in Figure 1.1(a). Relation* $\mathbf{E}$ *records employees with name* n *working for department* d*. For instance, tuple* e_1 *records that employee Amber works for department* 1*. Similarly, relation* $\mathbf{M}$ *records managers with name* mgr *responsible for department* dep*. For instance, tuple* m_1 *records that manager Xavier is responsible for department* 1*.*

E

	n	d
e_1	Amber	1
e_2	Billy	2
e_3	Chelsea	1

M

	mgr	dep
m_1	Xavier	1
m_2	Yvonne	2

(a) non-temporal Relations

Z

	mgr	$count$
z_1	Xavier	2
z_2	Yvonne	1

(b) Result

Figure 1.1: Non-Temporal Example.

To retrieve the number of employees each manager is responsible for, we formulate a query Q *that first joins relations* $\mathbf{E}$ *and* $\mathbf{M}$ *on the department attribute, and second groups the intermediate result by the attribute-values of* mgr *and aggregates each group using the aggregation function* $count$*. In all probability, query* Q *is formulated as follows:*

```
SELECT mgr, count(*)
FROM M JOIN E ON M.dep = E.d
GROUP BY mgr;
```

The result relation $\mathbf{Z}$ *of query* Q *is shown in Figure 1.1(b). Employees Amber (*e_1*) and Chelsea (*e_3*) work for department* 1*, thus manager Xavier (*m_1*) has an employee count of two which is represented by the result tuple* z_1*. Employee Billy (*e_2*) works for department* 2*, so manager Yvonne (*m_2*) has an employee count of one which is shown in the second result tuple* z_2*.*

In Example 1 we only store the latest snapshot of a modelled reality. For all practical purposes, it is often desired to know the history of a modelled reality. The people in a company may quit working, new employees or managers start working for the company or they could probably change the departments during time. To keep track of such changes, the recorded facts in a database are timestamped with intervals to indicate their valid time, i.e., the time periods these facts hold true.

Example 2 *In our next example, we extend both of the sample relations from the first example with interval timestamps. For each tuple, the recorded fact is true over its time interval* $[ts, te)$*, where* ts *is the inclusive start point and* te *the exclusive end point. For instance, as it is visible in tuple* e_3*, Chelsea is working at department* 1 *from the beginning of July until the end of November. The extended relations are shown in Figure 1.2 and as you can see, we have added one new tuple to each of the the relations* $\mathbf{E}$ *and* $\mathbf{M}$*. The newly added tuple* e_4 *in the employee relation represents in connection with tuple* e_1 *the fact that Amber switches departments at the last third of the year. On the other hand, we have added tuple* m_3 *to the manager relation which together with tuple* m_2*, describes a change of managers in department* 2 *after the first quarter.*

E

	n	d	ts	te
e_1	Amber	1	2012/01/01	2012/09/01
e_2	Billy	2	2012/01/01	2013/01/01
e_3	Chelsea	1	2012/07/01	2012/12/01
e_4	Amber	2	2012/09/01	2013/01/01

M

	mgr	dep	ts	te
m_1	Xavier	1	2012/01/01	2013/01/01
m_2	Yvonne	2	2012/01/01	2012/04/01
m_3	Zoe	2	2012/04/01	2013/01/01

Figure 1.2: Temporal Relations

By time-stamping the tuples of our relations, we can now express questions about the changes of facts in our sample database. Accordingly, we can run a temporal counterpart of query Q*, from Example 1, over the temporal relations. This means in detail, that it is possible to determine the number of employees each manager has at each point in time. To mark our new query* Q^T *as a temporal query, we prepend a fixed text string* `SEQVT` *as statement modifier for sequenced queries to our query* Q *from the previous example.*

```
SEQVT
SELECT mgr, count(*)
FROM M JOIN E ON M.dep = E.d
GROUP BY mgr;
```

For better a visualization, we illustrate the relations graphically in Figure 1.3. The tuples from the relations $\mathbf{E}$ *and* $\mathbf{M}$ *are drawn as horizontal lines above the underlying time line and have its start and end points set accordingly to their valid time intervals. Taking the change of managers in department* 2 *as an example, the horizontal line of Yvonne (*m_2*) is drawn from January until the end of March and with the beginning of April the horizontal line of Zoe (*m_3*) starts and proceeds until the end of December.*

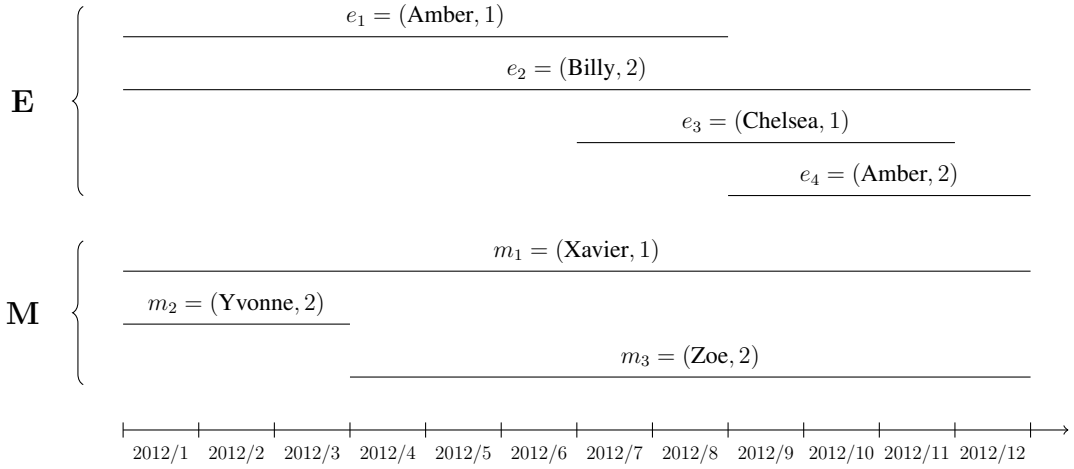


Figure 1.3: Graphical Representation of the Temporal Relations

Having in Figure 1.3 the graphical representation of our example relations at hand, we might easily derive the result of the temporal query Q^T*. To get the number of employees per manager at each point in time, we have to look up the time intervals of the managers. For each of them we check which and how many time intervals of its associated employees overlap this managers time interval. For example Zoe (*m_2*), which is the manager of department* 2 *since April, has one employee (Billy:* e_1*) from the beginning of April until the end of August. After that, Amber moves to Zoes department (*e_4*), so that Zoe is then supervising two employees from the beginning of September until the end of the year. This two facts are recorded separately as the tuples* z_5 *and* z_6 *and they are shown in Figure 1.4 where the whole result of query* Q^T *is stored in the temporal relation* Z*.*

Z

	mgr	*count*	*ts*	*te*
z_1	Xavier	1	2012/01/01	2012/07/01
z_2	Xavier	2	2012/07/01	2012/09/01
z_3	Xavier	1	2012/09/01	2012/12/01
z_4	Yvonne	1	2012/01/01	2012/04/01
z_5	Zoe	1	2012/04/01	2012/09/01
z_6	Zoe	2	2012/09/01	2013/01/01

Figure 1.4: Result of the Temporal Query Q^T.

The contributions of this thesis are as follows:

- I show a proposal of an SQL language using temporal statement modifiers for the querying of temporal data.
- I give a mechanism to compile queries with statement modifiers to statements with temporal primitives ALIGN and NORMALIZE.
- I show how the mechanism can be implemented by changing only the parser of the DBMs and leveraging the implementation of temporal primitives.
- I show and implementation into the database management system PostgreSQL.

The rest of this thesis is organized as follows. Chapter 2 discusses the reduction rules as well as the alignment operator, the normalization operator and some special cases. In chapter 3 we propose the definition of an SQL language with statement modifiers. Chapter 4 shows the mapping of queries with statement modifiers to queries with temporal primitives in algebraic expressions. Furthermore we propose the main idea how the mapping is basically implemented in the PostgreSQL parser. After that, Chapter 5 gives some insights about the implementation, clarifies some concepts of the translation algorithm proposed in Chapter 4 and points out some particular challenges.

2 Background and Related Work

2.1 General Preliminaries

In the scope of this project, we assume a linearly ordered, discrete time domain, Ω^T. A time interval is represented as a pair $T = [ts, te) \in \Omega^T \times \Omega^T$ and is a contiguous set of time points, where ts is the inclusive start point and te is the exclusive end point. The time interval that is associated with each tuple represents the tuple's valid time for which the recorded fact is true in the modeled reality. The schema of a temporal relation is denoted as $R = (A_1, ..., A_m, T)$, where the non-temporal attributes are represented as $A_1, ..., A_m$ and the time interval is represented as T. To shorten notations we use the abbreviation $A = A_1, ..., A_m$ for all non-temporal attributes of a relation. For $r.A_1 = s.A_1 \wedge ... \wedge r.A_m = s.A_m$ we write the abbreviated version $r.A = s.A$ and similarly for $r.ts = s.ts \wedge r.te = s.te$ we use the abbreviation $r.T = s.T$.

We write Q_{ra} to denote an SQL query Q in equivalent relational algebra. For instance given SQL query Q from Example 1 Q_{ra} is as follows:

$$Q_{ra} \quad = \quad {}_{mgr}\vartheta_{count(*)}(\mathbf{M} \bowtie_{\mathbf{M}.dep=\mathbf{E}.d} \mathbf{E})$$

Similar, for temporal SQL queries Q^T we use Q^T_{ra} to denote its equivalent temporal relational algebra expression, and we use ^{T} to indicate a temporal relational algebra operator, i.e., ϑ^T is the temporal aggregation operator and $\bowtie^T$ the temporal join operation. The relational algebra expression Q^T_{ra} equivalent to the temporal SQL query Q^T of Example 2 is as follows:

$$Q^T_{ra} \quad = \quad {}_{mgr}\vartheta^T_{count(*)}(\mathbf{M} \bowtie^T_{\mathbf{M}.dep=\mathbf{E}.d} \mathbf{E})$$

Besides the temporal operators ϑ^T and $\bowtie^T$, the other operators of the temporal relational algebra are selection σ^T, projection π^T, difference $-^T$, union $\cup^T$, intersection $\cap^T$, Cartesian product $\times^T$, left outer join $\tilde{\bowtie}^T$, right outer join $\tilde{\bowtie}^T$, full outer join $\tilde{\bowtie}^T$, and antijoin $\triangleright^T$. For the temporal projection $\pi^T_B(r)$ and the temporal aggregation ${}_B\vartheta^T_F(r)$ we require $B \subseteq A$. To denote a general relational algebra operator we use ψ, and ψ^T for its corresponding operator in the temporal relational algebra.

2.2 Reduction Rules

Dignös et al. [1] introduced a mechanism for the reduction of temporal operators to non-temporal operators with the use of temporal primitives, and we will use this reduction rules for the mapping of statement modifiers later on in this thesis. The main intuition behind the

temporal primitives is that they transform two relations with arbitrarily overlapping intervals into two relation where equality can be used to find overlapping intervals. The primitives and their parameters are specific for temporal operators and allow to use non-temporal operators with equality on timestamps after the primitive has been applied. The rules on how temporal primitives are used to reduce temporal operations are defined by so-called reduction rules shown in Table 2.1. More specifically the temporal primitives are the temporal alignment operator Φ and the temporal normalization operator $\mathcal{N}$. Additionally the so called absorb operator α is needed after joining the results of temporal alignments.

Operator			**Reduction**
Selection	$\sigma^T_\theta(r)$	$=$	$\sigma_\theta(r)$
Projection	$\pi^T_B(r)$	$=$	$\pi_{B,T}(\mathcal{N}_B(r;r))$
Aggregation	${}_B\vartheta^T_F(r)$	$=$	${}_{B,T}\vartheta_F(\mathcal{N}_A(r;r))$
Difference	$r -^T s$	$=$	$\mathcal{N}_A(r;s) - \mathcal{N}_A(s;r)$
Union	$r \cup^T s$	$=$	$\mathcal{N}_A(r;s) \cup \mathcal{N}_A(s;r)$
Intersection	$r \cap^T s$	$=$	$\mathcal{N}_A(r;s) \cap \mathcal{N}_A(s;r)$
Cart. Prod.	$r \times^T_\theta s$	$=$	$\alpha((r\Phi_{true}s) \bowtie_{r.T=s.T} (s\Phi_{true}r))$
Inner Join	$r \bowtie^T_\theta s$	$=$	$\alpha((r\Phi_\theta s) \bowtie_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
Left O. Join	$r ⟕^T_\theta s$	$=$	$\alpha((r\Phi_\theta s) ⟕_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
Right O. Join	$r ⟖^T_\theta s$	$=$	$\alpha((r\Phi_\theta s) ⟖_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
Full O. Join	$r ⟗^T_\theta s$	$=$	$\alpha((r\Phi_\theta s) ⟗_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
Anti Join	$r \triangleright^T_\theta s$	$=$	$(r\Phi_\theta s) \triangleright_{\theta \wedge r.T=s.T} (s\Phi_\theta r)$

Table 2.1: Reduction Rules

We now proceed by illustrating the reduction rules on our running example query Q^T_{ra}. For a better visualization we may split up the query into two parts, the temporal inner join $\bowtie^T$ and the temporal aggregation ϑ^T. By doing so, we get the following two algebraic expressions $Q^T_{ra_1}$ and $Q^T_{ra_2}$ where $\mathbf{X}$ stores the result relation of query $Q^T_{ra_1}$:

$$\begin{aligned} Q^T_{ra_1} &= \mathbf{X} \leftarrow \mathbf{M} \bowtie^T_{\mathbf{M}.dep=\mathbf{E}.d} \mathbf{E} \\ Q^T_{ra_2} &= {}_{mgr}\vartheta^T_{count(*)}(\mathbf{X}) \end{aligned}$$

By applying the reduction rules of Table 2.1 to the two temporal queries $Q^T_{ra_1}$ and $Q^T_{ra_2}$, we get the two reduced expressions $Q^T_{ra_{red1}}$ and $Q^T_{ra_{red2}}$, respectively. As before we label the result of query $Q^T_{ra_{red1}}$ as result relation X:

$$\begin{aligned} Q^T_{ra_{red1}} &= \mathbf{X} \leftarrow \alpha((\mathbf{M}\Phi_{\mathbf{M}.dep=\mathbf{E}.d}\mathbf{E}) \bowtie_{\mathbf{M}.dep=\mathbf{E}.d \wedge \mathbf{M}.T=\mathbf{E}.T} (\mathbf{E}\Phi_{\mathbf{M}.dep=\mathbf{E}.d}\mathbf{M})) \\ Q^T_{ra_{red2}} &= {}_{mgr,T}\vartheta_{count(*)}(\mathcal{N}_{mgr}(\mathbf{X};\mathbf{X})) \end{aligned}$$

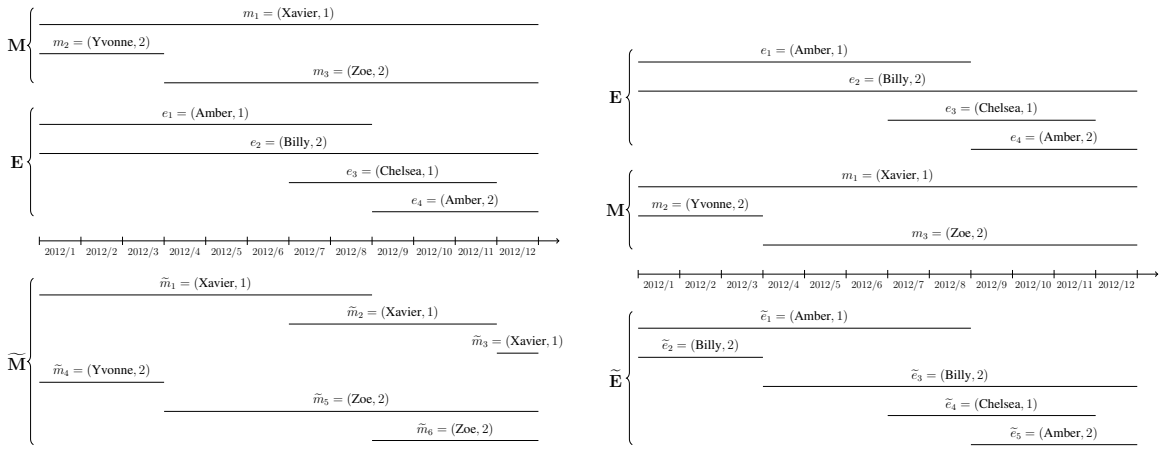
Note that $Q^T_{ra_{red1}}$ and $Q^T_{ra_{red2}}$ due to the reduction, now do not contain temporal operators, only the non-temporal operators join $\bowtie$ and aggregation ϑ, and temporal primitives. The next sections of this chapter investigate the temporal primitives $r\Phi_\theta s$ and $\mathcal{N}_B(r;s)$, where for the former primitive we will also quickly explain the absorb operator α.

2.3 The Alignment Operator

In our first reduced relational algebra expression $Q^T_{ra_{red1}}$ the temporal alignment occurs twice. The first time on the left side of the join as $\mathbf{M}\Phi_{\mathbf{M}.dep=\mathbf{E}.d}\mathbf{E}$ and the second time similarly on the right side of the join as $\mathbf{E}\Phi_{\mathbf{M}.dep=\mathbf{E}.d}\mathbf{M}$. The theta condition $\theta \equiv \mathbf{M}.dep = \mathbf{E}.d$ is the same for both alignment operations, but the inverted input arguments relation $\mathbf{M}$ and $\mathbf{E}$ indicate the difference between them. One time it means that $\mathbf{M}$ is aligned with respect to $\mathbf{E}$ and the second time that $\mathbf{E}$ is aligned with respect to $\mathbf{M}$. For the first occurrence it says in detail that for each tuple in $\mathbf{M}$, we search all tuples in $\mathbf{E}$ that intersect with its time interval and simultaneously satisfy the theta condition $\theta \equiv \mathbf{M}.dep = \mathbf{E}.d$. For each match we get a result tuple with the non-temporal attributes of $\mathbf{M}$ and an adjusted time interval $[ts, te)$ resulting from the intersection of the intervals. Each time gap of an $\mathbf{M}$-tuple that is not covered by an appropriate $\mathbf{E}$-tuple is also producing a result-tuple, having its timestamp interval $[ts, te)$ set to the start and end point of the uncovered gap.

Example 3 *The alignment of* $\mathbf{M}\Phi_{\mathbf{M}.dep=\mathbf{E}.d}\mathbf{E}$ *is shown in Figure 2.1(a). For instance, the first result tuple* $\widetilde{m}_1 = (Xavier, 1, [2012/1/1, 2012/9/1))$ *is derived from* m_1 *and* e_1*, which are fulfilling the theta condition* $\mathbf{M}.dep = \mathbf{E}.d$ *and having an intersection of their time intervals. The second result tuple* $\widetilde{m}_2 = (Xavier, 1, [2012/9/1, 2012/12/1))$ *is derived from* m_1 *and* e_2 *and their time intersections. The third result tuple is derived from* m_1 *and its time interval [2012/12/1, 2013/1/1) which has no corresponding match in any employee-tuple from relation* $\mathbf{E}$ *that fulfills the theta condition* θ*.*

The second alignment $\mathbf{E}\Phi_{\mathbf{M}.dep=\mathbf{E}.d}\mathbf{M}$ *is shown in Figure 2.1(b). Keep in mind that this time, the argument relations are swapped so that the relation* $\mathbf{E}$ *is aligned with respect to* $\mathbf{M}$*. The theta condition* θ *in contrast stays the same. For instance, the second result tuple* $\widetilde{e}_2 = (Billy, 2, [2012/1/1, 2012/4/1))$ *is derived from the tuples* e_2 *and* m_2 *and the third result tuple* $\widetilde{e}_3 = (Billy, 2, [2012/4/1, 2013/1/1))$ *results from* e_2 *and the intersection of its time interval and the time interval of* m_3*.*



(a) Alignment of $\mathbf{M}$ (b) Alignment of $\mathbf{E}$

Figure 2.1: Temporal Alignments

We can see that now in the place where before we had arbitrary overlapping time intervals for the join, we now have equal timestamps. For instance, tuple m_2 and e_2 are overlapping and satisfy the join condition θ, after alignment we have tuple $\widetilde{m}_4$ and $\widetilde{e}_2$ with equal interval timestamps corresponding to the common interval timestamps of m_2 and e_2.

For all of the temporal operators $\psi^T \in \{\times^T, \bowtie^T, \rtimes^T, \ltimes^T, \rtimes\ltimes^T, \triangleright^T\}$ the temporal alignment operator is used to adjust the timestamps of the argument relations as shown in Example 3. After that the corresponding non-temporal operator ψ is applied in order to join the adjusted relations together. Hence we have the desired equal timestamps now, we can extend the qualifiers on joins with an equality comparison between interval timestamps. Example 4 shows a sample of an inner join on the previously aligned relations.

Example 4 *After processing the alignment operations and therefore having the aligned relations $\widetilde{\mathbf{M}}$ and $\widetilde{\mathbf{E}}$ at hand, we are now able to perform the actual inner join. As the reduction rule for inner joins in Table 2.1 have the form $\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) \bowtie_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$, the join conditions for our example turn out to be a conjunction of the well known theta condition $\theta \equiv (\mathbf{M}.dep = \mathbf{E}.d)$ and equality comparison of the interval timestamps from $\widetilde{\mathbf{M}}$ and $\widetilde{\mathbf{E}}$, so in this case: $\widetilde{\mathbf{M}}.T = \widetilde{\mathbf{E}}.T$. The resulting relation $\mathbf{X}$ after the inner join is shown graphically in Figure 2.2.*

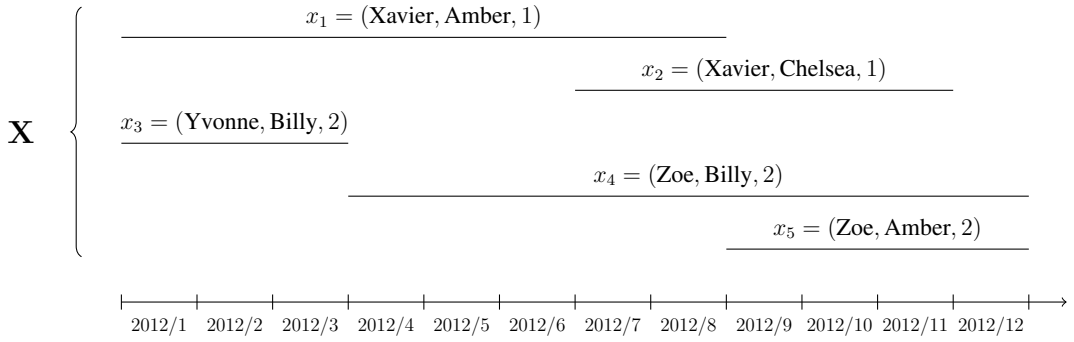


Figure 2.2: Result after Inner Join

The absorb operator $\alpha(\mathbf{r})$ has to be performed as last operation of the reduction rules of any temporal join (except the anti join). As proposed by Dignös et al. [1], the operator eliminates temporal duplicates over non-maximal time intervals. That means that the operator searches for tuples from the argument relation that have equal non-temporal attributes. If one of those tuples time interval is a proper subset of another ones interval, we remove it from the argument relation. Applying this operator to our result relation $\mathbf{X}$ does no lead to any changes, since such temporal duplicates do not exist in our example.

2.4 The Normalization Operator

The second part of our temporal Query is $Q^T_{ra_{red2}} = {}_{mgr,T}\vartheta_{count(*)}(\mathcal{N}_{mgr}(\mathbf{X};\mathbf{X}))$, thus the temporal normalization operator $\mathcal{N}_A(\mathbf{r};\mathbf{r})$ is the next temporal primitive that has to be processed in our example. The expression $\mathcal{N}_{mgr}(\mathbf{X};\mathbf{X})$ indicates that the relation $\mathbf{X}$ will be normalized with respect to itself $\mathbf{X}$ and the Attribute mgr. That means in detail that we get the normalized result tuples if we adjust the intervals of each tuple in $\mathbf{X}$ by the start and end point of all the tuples of $\mathbf{X}$ in the same group. In our example, $\mathbf{X}$ is grouped by the attribute mgr into the three groups with $mgr = Xavier$, $mgr = Yvonne$ and $mgr = Zoe$. Similarly to the alignment operator, each gap in the interval of a $\mathbf{X}$-tuple, that is not covered by any $\mathbf{X}$-tuple in the same group, produces a result-tuple as well.

Example 5 *The normalization of the relation* $\mathbf{X}$ *with respect to itself, having the attribute* mgr *to be used for grouping, is shown in Figure 2.3. For instance, the first result tuple* $y_1 = (Xavier, Amber, 1, [2012/1/1, 2012/7/1))$ *and the the second result tuple* $y_2 = (Xavier, Amber, 1, [2012/7/1, 2012/9/1))$ *are derived from* x_1 *which is split up by the start point* $ts = 2012/7/1$ *of* x_2 *since* x_1 *and* x_2 *have both the same value* $Xavier$ *in their* mgr *attribute. Likewise, the third result tuple* $y_3 = (Xavier, Chelsea, 1, [2012/7/1, 2012/9/1))$ *as well as the fourth result tuple* $y_4 = (Xavier, Chelsea, 1, [2012/9/1, 2012/12/1))$ *are derived from* x_2 *which is split up by the end point* $te = 2012/9/1$ *of* x_1.

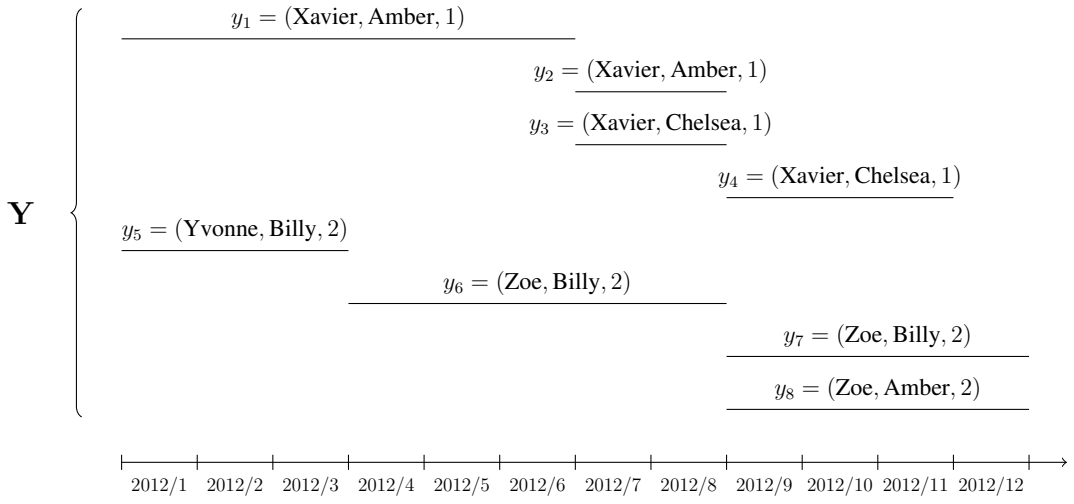


Figure 2.3: Temporal Normalization

After the temporal normalization, it is now possible to make the non-temporal aggregation ${}_{mgr,T}\vartheta_{count(*)}Y$ *over the result relation* $\mathbf{Y}$*, we can use* T *in the grouping, since interval timestamps belonging to the same group* mgr *are now either equal or disjoint. Figure 2.4 shows the result of this aggregation, which is equal to the assumed result of the query* $Q^T = {}_{mgr}\vartheta^T_{count(*)}(\mathbf{M} \bowtie^T_{\mathbf{M}.dep=\mathbf{E}.d} \mathbf{E})$ *shown in Figure 1.4.*

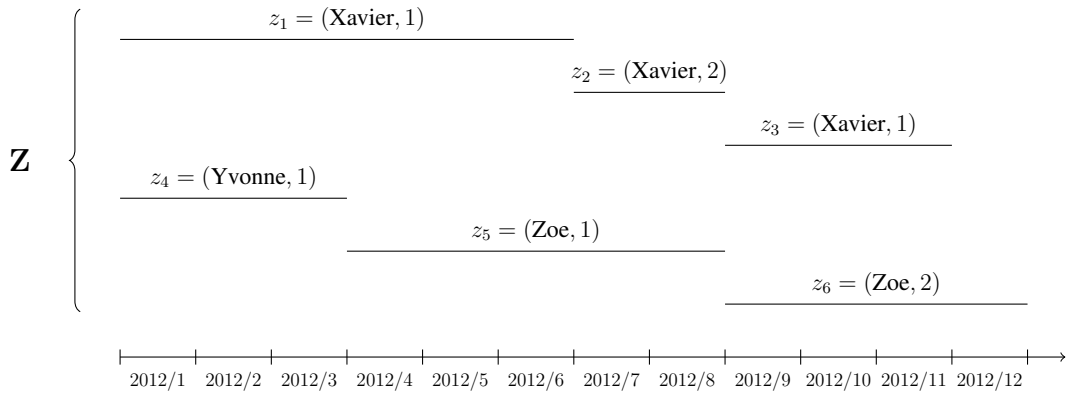


Figure 2.4: Aggregation

2.5 Timestamp Propagation

As previously described, it is sometimes necessary to propagate timestamps when reduction rules are applied. As explained by Dignös et al. [1], this has to be done when interval timestamps are used in predicates and functions of temporal operators ψ^T. If such a case occurs, we apply the timestamp propagating operator $\epsilon_U(r)$ by propagating interval timestamps as non-temporal attributes. That means that we add attributes with name $U = [us, ue)$ to the schema of the argument relation which are identical to the interval timestamps $T = [ts, te)$. The temporal primitives won't adjust the non-temporal interval U so that U stores the original interval timestamps and can be used instead of T in the predicates and/or functions.

Example 6 *We do not use predicates or function on timestamps in our running example, thus timestamp propagation is not required. But to illustrate the behavior of the timestamp propagation operator, we show in Figure 2.5 with* $\epsilon_U(\mathrm{M})$ *the propagation of the timestamps for the manager relation* M*. For instance, when we are applying the normalization primitive for a subsequent aggregation on the relation* M*, we can still use the original timestamp* U *(that is not adjusted) in aggregation functions to, e.g., get the average duration of manager contracts.*

$\mathbf{M}$

	mgr	*dep*	*us*	*ue*	*ts*	*te*
m_1	Xavier	1	2012/01/01	2013/01/01	2012/01/01	2013/01/01
m_2	Yvonne	2	2012/01/01	2012/04/01	2012/01/01	2012/04/01
m_3	Zoe	2	2012/04/01	2013/01/01	2012/04/01	2013/01/01

Figure 2.5: Timestamp Propagation of Relation M

3 Definition of an SQL Language with Statement Modifiers

This chapter introduces an SQL language that supports temporal statement modifiers and it describes how statement modifiers can still allow to scale attribute values at query time. In the first section we will look at the basic definition of a simple select statement of the PostgreSQL parser [3] and how it has to be modified in order to support temporal statement modifiers. In the second section we will determine the scope of such modifiers and show the advantages as well as the limitations that come along with our chosen scope. After that we will show why and how the support for set operations with statement modifiers differs from the way that statement modifiers are supported for usual select statements. In the fourth section we will describe some additional parameters for statement modifiers and we will define an appropriate grammar for the SQL language. In the last section we will have a look at the scaling of attribute values. We will show the need of such scaling functions and we will describe how the scaling is still allowed in our new modified SQL.

3.1 Basic Definition for Temporal Statement Modifiers

Figure 3.1 shows the form of a general and well known SQL select statement. As for natural languages, SQL has a grammar that defines the language in terms of syntax, i.e., determines whether a query is syntactically correct. The syntax of the SQL language in PostgeSQL is ensured by defining rules in the grammar-file gram.y (src/backend/parser/gram.y[1]). The grammar is highly similar to the Extended Backus–Naur Form syntax and thus can be read likewise. Terminal symbols are written in CAPITALS and non-capitals are used for non-terminals, i.e., recursive rules. A simplified version of the basic grammar rule for a select statement is shown on the right side in Figure 3.1. Curly brackets enclose c-code which is performed if a grammar rule holds true. The c-code of each grammar rule is used to build up a raw parse-tree and in this case it is used to create a c-struct representing a select statement. The c-code is omitted in the figure.

One of the main tasks within this project is to integrate statement modifiers into the PostgreSQL parser, so that it is possible do distinguish which statements need to be translated to SQL queries with temporal primitives. Böhlen et al. [2] introduced such temporal statement modifiers to provide temporal support to an existing query language. Statement modifiers are designed to be prepended to queries, and a whole grammar that supports many temporal features has been proposed. In the case of the reduction rules, we are dealing with sequenced

[1] http://www.ifi.uzh.ch/dbtg/research/align.html

```
                          simple_select:
SELECT clause                 SELECT target_list
FROM clause                   from_clause
WHERE clause                  where_clause
GROUP clause                  group_clause
HAVING clause                 having_clause
                              {
                               // create select statement
                              }
```

Figure 3.1: SQL (l.) and Grammar Definition in PostreSQL (r.) of common Select Statements.

queries and valid time timestamps so we take as temporal statement modifier the fixed text string `SEQVT` as already denoted in Chapter 1. Prepending this modifier to a query lets us know that the statement has to be processed with a modified behavior, that means in our case, to apply reduction rules. To define it appropriately in the grammar, the additional non-terminal `modifiers` has to be added to the simple-select rule as well as an appropriate rule to this non-terminal. The newly created rule means nothing more than returning false or true to the appropriate attribute of the select statement. The modified SQL and its grammar are shown in Figure 3.2.

```
                          simple_select:
SEQVT                         modifiers
SELECT clause                 SELECT target_list
FROM clause                   from_clause
WHERE clause                  where_clause
GROUP clause                  group_clause
HAVING clause                 having_clause
                              {
                               // create select statement and
                               // mark it appropriate if seqvt
                              }

                          modifiers:
                              SEQVT {
                               // return some true value
                              }
                              | /*EMPTY*/ {
                               // return some false value
                              }
```

Figure 3.2: SQL (l.) and Grammar (r.) of Select Statements with a Statement Modifier.

3.2 Scope of Statement Modifiers

An important aspect of temporal statement modifiers is to define their scope. There are basically two variants. Either the statement modifier affects only the statement at the current level, or it affects also the deeper nested statements like query expressions inside the from clause. Assume we have a statement we want to be processed in a temporal manner, but the statement itself has another statement in its from clause that we want to be processed in the usual non-temporal manner. If we decide for the latter of the above mentioned variants, we wouldn't have the possibility to query usual non-temporal select statements inside the from clause and therefore patronize the user and downgrade the user experience. One alternative to avoid such a problem would be to create a second statement modifier like `NOSEQVT`. This modifier could be used to cancel the handed down temporal behavior from on the statement it is attached to, see left side in Figure 3.3. Besides that the implementation effort would increase with this approach it seems a bit cumbersome to mark select statements only that they are processed in their usual non-temporal behaviour. The basic principle of statement modifiers would be missed since they should generally be used to modify the ordinary behaviour of statements [2] rather than countermand a modified behaviour. On these grounds we choose to stay with the non-inheritable Modifier as it is shown on the right side of Figure 3.3.

```
SEQVT SELECT count(*)
      FROM (
           NOSEQVT SELECT mgr
                   FROM M
           ) m;
```

```
SEQVT SELECT count(*)
      FROM (
           SELECT mgr
           FROM M
           ) m;
```

Figure 3.3: Additional Modifier (l.) vs. non-inheritable Modifier (r.)

3.3 Supporting Statement Modifiers for Set Operations

Next we need to determine how temporal statement modifiers may support the set operations difference $r - s$, union $r \cup s$ and intersection $r \cap s$ properly. A set operation has two input arguments, usually select statements to the left and the right of the operation. Until now, the modified grammar allows us to prepend statement modifiers to both of the select statements. In Figure 3.4 we see the approximate form of SQL queries for set operations that have statement modifiers prepended in front of their select statements.

```
SEQVT SELECT clause FROM clause ...
[UNION|INTERSECT|EXCEPT]
SEQVT SELECT clause FROM clause ...
```

Figure 3.4: Wrong SQL for Statement Modifiers prepended to Set Operations.

Unfortunately, such a query is not sufficient enough to mark the set operation appropriately so that it is treated as a temporal operation. Instead each select statement is processed separately in a temporal manner and after that the set operation itself is still processed in a non-temporal manner.

In order to achieve our desired temporal behavior for set operations we need to modify the PostgreSQL grammar again. If we think back to the reduction rules in Table 2.1 we see that the set operations must be affected on the whole by a single statement modifier. One might also still remember that in SQL a set operation is said to be one whole query expression. That is one reason why the PostgreSQL grammar defines that a select statement is either one single select statement, or in case of set operations, it is a tree of select statements. Assuming a `UNION` operation with two select statements as arguments, the `UNION` operator would be a select statement itself as the root node in the tree. It would have set an attribute to a value which identifies itself as a `UNION` operation. Additionally this root select statement would have set the two argument select statements as child nodes. Looking at Figure 3.5 it is clear that we only have to attach the temporal statement modifier to the root select statement, so that the temporal query can be appropriately translated to its query with temporal primitives.

Figure 3.5: Tree of Select Statements

To distinguish in SQL that the temporal statement modifier should be applied to the set operation and not to the first argument of the operation, we need to make use of parentheses. After the statement modifier we would write a left parenthesis followed by the set operation and conclude the expression with a right parenthesis. Figure 3.6 shows the form of such an SQL expression.

```
SEQVT (
    SELECT clause FROM clause ...
    [UNION|INTERSECT|EXCEPT]
    SELECT clause FROM clause ...
)
```

Figure 3.6: Correct SQL for Statement Modifiers prepended to Set Operations.

To integrate this into the PostgreSQL grammar, we need to modify the rules for set operations. Currently this rules are alternative rules for the `simple_select` non-terminal. In Figure 3.7 we see the simplified versions of the original set operation rules on the left side. We now have to replace this three rules with a reference to the new non-terminal `set_operation_modifiers`. This non-terminal contains two different rules. In the first rule it checks whether a temporal statement modifier is set and if the set operation expression

is properly embraced by parentheses. The second rule is needed if no statement modifier is at hand and nothing special has to be done. Both rules hold a reference to a second new non-terminal `set_operation`. As the name denotes, the `set_operation`-grammar contains the original three set operation rules.

```
simple_select:
    ...
    | select_clause UNION select_clause
    {
      // create select statement,
      // id it as UNION statement
      // and add child statements
    }
    | select_clause INTERSECT select_clause
    {
      // create select statement,
      // id it as INTERSECT statement
      // and add child statements
    }
    | select_clause EXCEPT select_clause
    {
      // create select statement,
      // id it as EXCEPT statement
      // and add child statements
    }
    ...
```

```
simple_select:
    ...
    | set_operation_modifiers
    {
      // return set operation including
      // the statement modifier if needed
    }
    ...
  ;

set_operation_modifiers:
    modifiers '(' set_operation ')'
    {
      // append the statement modifier
      // to the set operation statement
    }
    | set_operation
    {
      // return just the set operation
    }
  ;

set_operation:
    select_clause UNION select_clause
    {
      // original behavior
    }
    | select_clause INTERSECT select_clause
    {
      // original behavior
    }
    | select_clause EXCEPT select_clause
    {
      // original behavior
    }
  ;
```

Figure 3.7: Original (l.) vs. Modified (r.) Grammar for Set Operations

At the moment, the `set_operation_modifiers:`-grammar in combination with the `modifiers:`-grammar will produce some so called shift/reduce-conflicts. Such conflicts occur when the parser has two or more possible ways to traverse through the grammar verifying the syntax of a statement as correct. To visualize the problem for our approach we show two different ways to parse a set operation that is embraced with parentheses and has no keyword `SEQVT` prepended to it, e.g., the statement `(SELECT * FROM M UNION SELECT * FROM E)`. The two traversals are shown in Figure 3.8. On the right side we can parse the statement by following the rules `SelectStmt`, `select_no_parens`, `simple_select`, `set_operation_modifiers`, `modifiers '(' set_operation ')'`, `/* EMPTY */`. On the left side we can parse the statement differently by following the rules `SelectStmt`, `select_with_parens`, `'(' select_no_parens ')'`, `simple_select`, `set_operation_modifiers`, `set_operation`.

```
SelectStmt:
    select_no_parens
    ...
  ;

select_no_parens:
    simple_select
    ...
  ;

simple_select:
    ...
    | set_operation_modifiers
    ...
  ;

set_operation_modifiers:
    modifiers '(' set_operation ')'
    ...
  ;

modifiers:
    ...
    | /*EMPTY*/
    ...
  ;
```

```
SelectStmt:
    ...
    | select_with_parens
    ...
  ;

select_with_parens:
    '(' select_no_parens ')'
    ...
  ;

select_no_parens:
    simple_select
    ...
  ;

simple_select:
    ...
    | set_operation_modifiers
    ...
  ;

set_operation_modifiers:
    ...
    | set_operation
    ...
  ;
```

Figure 3.8: Shift/Reduce-Conflict for non-temporal Set Operations

To get rid of the errors we have to delete the alternative rule `/*EMPTY*/` in the `modifiers:`-grammar. After that we have to give back the possibility that common select statements may exist without statement modifiers. We solve this by adding the original rule for usual select statements as an alternative to our modified rule. The changes that are needed in the grammar are shown in Figure 3.9.

```
simple_select:
    modifiers
    SELECT target_list from_clause
    where_clause group_clause having_clause
    {
     // create select statement with statement modifier
    }
    | SELECT target_list from_clause
    where_clause group_clause having_clause
    {
     // or create statement without statement modifier
    }
    ...
  ;
```

```
modifiers:
    SEQVT {
     // return some true value
    }
  ;
```

Figure 3.9: Changes needed to prevent Errors in the Grammar

3.4 Additional Parameters for Statement Modifiers

The translation of the queries with statement modifiers to queries with temporal primitives must be executed if the corresponding temporal statement modifier is prepended. Considering our running examples in Chapter 1 and the reduction rules in Chapter 2, it would be favorable to be able to specify the names of the timestamp attributes. Otherwise names of the timestamp attributes need to be hard coded into the translation algorithm of the parser and that would lead to a bad user experience. Either the relations which the user is using must have exactly the timestamp names we allow, or the user needs to perform a renaming for each relation he intends to use in a temporal query.

A way to improve this, is to pass along the names of the timestamp columns together with the temporal statement modifier. With that we are able to determine the timestamp columns of a statement when it comes to the translation algorithm. Applying this to our rule of the `modifiers` nonterminal leads to the conclusion that we rather have to create a statement modifier object than only setting a simple flag to true. The modifier object then holds the names of the timestamp columns as shown in the first `modifiers`-rule in Figure 3.10.

```
SEQVT ts te
SELECT clause
FROM clause
WHERE clause
GROUP clause
HAVING clause
```

```
simple_select:
    modifiers
    SELECT target_list
    from_clause
    where_clause
    group_clause
    having_clause
    {
      // create select statement and
      // append seqvt object if desired
    }
    ...
  ;

modifiers:
    SEQVT ColId ColId }
      // return object with the column names
    }
    | SEQVT {
      // return object with default names
    }
  ;
```

Figure 3.10: SQL (l.) and the Grammar (r.) for the extended Statement Modifier.

We can still define the grammar of the parser in such a way, that omitting the timestamp names is allowed and some default names, for example ts and te, are stored in the statement modifier object and will be used for the processing of the reduction rules. This alternative rule

is shown as the second `modifiers`-rule in Figure 3.10

Nevertheless, one could argue that it would be probably better if we are allowed to pass the names of each relations timestamps. On one side, the whole statement modifier extension would become very flexible for databases with relations having different timestamp names. On the other side, more complex queries which cover multiple relations, could become almost impossible to read and the implementation will surely become way harder to accomplish. So again, in terms of keeping things simple, we leave it the way that we pass only a single column name for the starting point of the time interval and a second column name for the end point of the interval. Besides favoring simplicity, these design decision would constrain the users to have a constant naming convention for columns that have the same duty, so in our case these are the interval timestamps columns.

3.5 Allowing the Scaling of Attribute Values

As described earlier, the alignment operator Φ and the normalization operator $\mathcal{N}$ adjust interval timestamps. The scaling of attribute values becomes necessary because not all attribute values may remain valid if the associated interval timestamps change due to the temporal adjustments [4]. Dignös et al. [5] have proposed how such a scaling can be supported in SQL at query time. The final step of defining the SQL language with temporal statement modifiers is to ensure, that the scaling of attribute values is still supported in queries with such statement modifiers.

Example 7 *Figure 3.11 shows an example for which the scaling of attribute values is required. Relation* $\mathbf{L}$ *in Figure 3.11(a) is the employee relation* $\mathbf{E}$ *with an additional column* l *that records the loan that each employee earns during the associated interval* $[ts, te)$*. For instance, tuple* l_1 *records that Amber works at department* 1 *from the beginning of the year until the end of August and earns in this period a total of* $244K$*. To query the time-varying sum of the employees loan we use the query* $\mathbf{Z} = \vartheta^T_{SUM(scale(l))}(\mathbf{L})$*. We have a temporal aggregation* ϑ^T *and the function* $scale(l)$ *inside the aggregation function* $SUM()$ *that indicates that the value of attribute* l *has to be scaled before the actual aggregation function is performed. The result of this query is shown in Figure 3.11(b).*

L

	n	d	l	ts	te
l_1	Amber	1	$244K$	2012/01/01	2012/09/01
l_2	Billy	2	$366K$	2012/01/01	2013/01/01
l_3	Chelsea	1	$184K$	2012/07/01	2012/12/01
l_4	Amber	2	$122K$	2012/09/01	2013/01/01

(a) New Temporal Relation $\mathbf{L}$

Z

	sum	ts	te
z_1	$364K$	2012/01/01	2012/07/01
z_2	$186K$	2012/07/01	2012/09/01
z_3	$273K$	2012/09/01	2012/12/01
z_4	$62K$	2012/12/01	2013/01/01

(b) $\vartheta^T_{SUM(scale(l))}(\mathbf{L})$

Figure 3.11: Scaling of Attribute Values.

With the graphical representation in Figure 3.12 it is easy to understand the results without knowing the exact mechanics of the scaling. We can see that each employee has a loan of $1K$

per day. Knowing that fact lets us easily derive the result tuples from relation **Z***. For instance the first result tuple* $z_1 = (364K, [2012/01/01, 2012/07/01)$ *is derived from the employee-loans of* l_1 *and* l_2*. For each employee we add up* $1K$ *per day until the beginning of July, when the new employees loan* l_3 *changes the sum per day leading to the starting point of the second result tuple* z_2*.*

To understand the detailed approach to get that result relation, the query needs to be reduced with respect to the reduction rule for temporal aggregations shown in Table 2.1. As it is explained in detail later on, also a timestamp propagation has to be performed on the employee relation **L**. Applying this two steps to our original query leads us to the expression $\mathbf{Z} = \vartheta_{SUM(scale(l))}(\mathcal{N}(\epsilon_U(\mathbf{L}); \epsilon_U(\mathbf{L}))$. After propagating the timestamps and having the relation **L** normalized with respect to itself, the original loan values must be scaled to the adjusted intervals of the normalized tuples. For the final step, the non-temporal aggregation function $\vartheta_{SUM()}$ needs to be performed to deliver us the time-varying sum of employee-loans as shown in Figure 3.11(b) and in Figure 3.12.

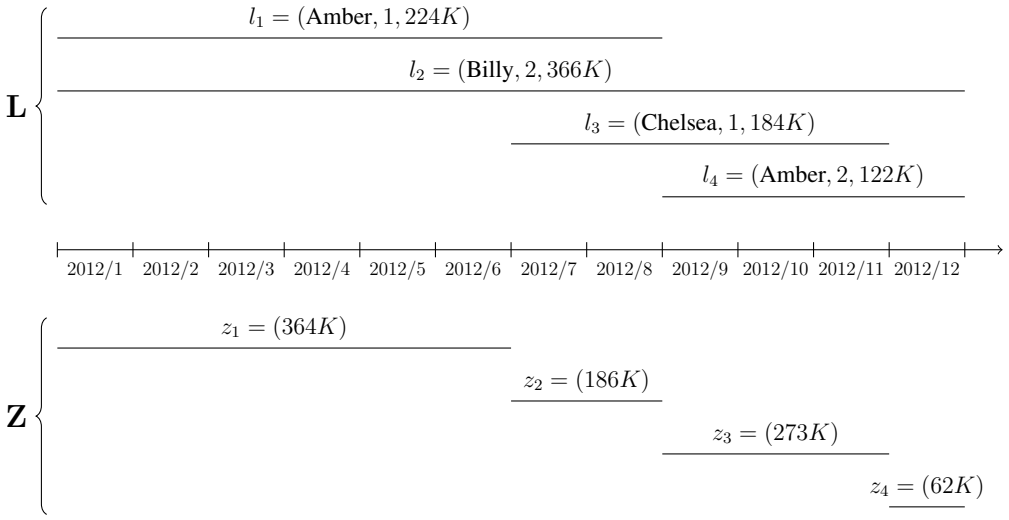


Figure 3.12: Graphical Representation of Scaling Attribute Values.

As proposed by Dignös et al. [5] there exist different possible scaling functions, namely *Uniform Scaling*, *Atomic Scaling* and *Trend Scaling*. All of them have in common that the scaling functions do not only need as input parameter the attribute that should be scaled. As already denoted in the example, it is also necessary that the adjusted timestamps as well as the original timestamps are passed as parameters to the function. So that is why all the scaling methods appear with a function header that have approximately a form that looks like $scale(x, Ts, Te, Us, Ue)$. As an example, we show a function in Figure 3.13 that scales attribute values proportionally to the length of the new adjusted timestamps. This *Uniform Scaling* is also the scaling function that is needed to get the appropriate results in our example above.

```
CREATE OR REPLACE FUNCTION
scaleU(x FLOAT, ts_new DATE, te_new DATE, ts_old DATE, te_old DATE)
RETURNS FLOAT AS $$
BEGIN
RETURN x * (te_new - ts_new) / (te_old - ts_old);
END; $$ LANGUAGE PLPGSQL;
```

Figure 3.13: Function for Uniform Scaling.

Hence the original timestamps still have to exist when the scaling takes place, it is necessary that we use timestamp propagation $\epsilon_U(\mathbf{L})$ before the temporal adjustment is done. Having the propagation and the normalization done, it is then possible to use the original timestamps us and ue to perform a valid scaling.

All in all we get a definite order of operations that has to be met so that the scaling of attribute values is done right. But all of this has to be taken care of by the translation algorithm that transforms queries with statement modifers to queries with temporal primitives. We will pursue this topic in the next chapter. As we can see, the grammar of the PostgreSQL parser itself does not need to be modified any further so that a scaling would be possible. The user itself has to take care that he passes the right timestamps to the scale-function. Regarding the scale function above, the second and third parameter should be filled with the timestamp names that are used for the interval timestamps in the database. For the forth and the fifth parameter, the user should pass along the names that were created in the course of the timestamp propagation.

4 The SQL mapping

In Chapter 4 we investigate the mapping of queries with statement modifiers to queries with temporal primitives. In the first section we will look at the already existing implementation of temporal primitives in PostgreSQL. We show the usage of this primitives in an example and we illustrate the advantages of `WITH`-clauses in this context. Then, in the second section, we will demonstrate the mapping of queries over algebra expressions and determine the actual order for processing temporal operations. In the third section we will explain the mapping in SQL and the basic approach of our implementation of the translation algorithm into the PostgreSQL parser.

4.1 Implementation of the Temporal Primitives

An implementation of the temporal primitives in the kernel of the PostgreSQL database system has already been described by Dignös et al. [1]. They've used a very straightforward approach so that the formulation of temporal SQL queries are relatively easy. Prerequisites are the knowledge of how to write the relational algebra expressions of the desired temporal queries and how the reduction rules need to be applied to reduce the expressions properly. With that we are able to assemble the corresponding SQL queries and statements containing the temporal primitives. The SQL syntax for these primitives is shown in Table 4.1.

$\epsilon_U(r)$:	`SELECT ts us, te ue, * FROM r`
$\mathcal{N}_B(r;s)$:	`FROM (r` **`NORMALIZE`** `s` **`USING(`**B**`)`** `) r`
$r\Phi_\theta s$:	`FROM (r` **`ALIGN`** `s` **`ON`** θ`) r`
$\alpha(r)$:	`SELECT` **`ABSORB`** `* FROM r`

Table 4.1: Implementation of Temporal Primitives

The timestamp propagation operation $\epsilon_U(r)$ can be achieved by using a common non-temporal select statement that projects the argument relation with all of its attributes and extends it with copies us and ue of the original timestamps ts and te as explicit non-temporal attributes. As previously specified, the original timestamps must be replaced with the references to the propagated attributes whenever the original timestamps are needed in theta conditions θ, in aggregate functions ϑ_F and in scaling functions. The normalization operator $\mathcal{N}_B(r;s)$ is made accessible by using the keyword `NORMALIZE` inside the `FROM`-clause. We define the primitive's corresponding grouping-clause by writing the grouping attributes inside

the brackets of `USING()`. To the immediate left and right side of the keyword `NORMALIZE` we write the two input argument relations so that the first relation is normalized with respect to the second one. In a similar manner we can handle the alignment operation $r\Phi_\theta s$ by using the keyword `ALIGN` instead of `NORMALIZE`. The associated theta-condition θ has to be formulated by starting with the keyword `ON` and is then followed by the actual condition(s). Finally, the absorb operator α is accessible by using the keyword `ABSORB`.

As we will also see it in Example 8 afterwards, `WITH`-clauses play an important part when it comes to the use of reduction rules and temporal primitives. Without using `WITH`-clauses, queries can become exponentially big. For instance, when we have to apply timestamp propagation to an argument relation r, we would be forced to rewrite the timestamp propagation `SELECT ts us, te ue, * FROM r` for each occurrence of this one argument relation. That is at least twice, since every reduction requires each argument relation twice. By using the `WITH`-clause it is only necessary to define such expressions once in the beginning of the temporal statement. Each occurrence of the original expression can then be substituted with the alias of the virtual table that has been created along with the `WITH`-clause, and thus queries grow only linearly instead of exponentially.

Example 8 *The affinity between the relational algebra and the SQL syntax can also be shown in a more complex example. Consider our relational algebra expressions $Q^T_{ra_{red1}}$ and $Q^T_{ra_{red2}}$ the corresponding SQL is shown on the left-hand side in Figure 4.1, where the `WITH` statement corresponds to $Q^T_{ra_{red1}}$ dealing with the temporal join, and the second expression $Q^T_{ra_{red2}}$ is the temporal aggregation corresponding to the remaining SQL. On the right-hand side in Figure 4.1 we can clearly see that the query is becoming much bigger when not taking advantage of the `WITH`-clause. Without the virtual table X from the `WITH`-clause we are forced to write the whole SQL query for the temporal inner join twice, once for the left and once for the right side of the `NORMALIZE` operation.*

```
WITH X AS (
    SELECT ABSORB mgr, n, M.ts, M.te
    FROM (M ALIGN E ON M.dep = E.d) M
         JOIN
         (E ALIGN M ON M.dep = E.d) E
         ON M.dep = E.d
         AND M.ts=E.ts AND M.te=E.te
)
SELECT mgr, count(*), ts, ts
FROM (X NORMALIZE X USING(mgr)) Y
GROUP BY mgr, ts, ts;
```

```
SELECT mgr, count(*), ts, ts
FROM (
    (SELECT ABSORB mgr, n, M.ts, M.te
    FROM (M ALIGN E ON M.dep = E.d) M
         JOIN
         (E ALIGN M ON M.dep = E.d) E
         ON M.dep = E.d
         AND M.ts=E.ts AND M.te=E.te) X1
    NORMALIZE
    (SELECT ABSORB mgr, n, M.ts, M.te
    FROM (M ALIGN E ON M.dep = E.d) M
         JOIN
         (E ALIGN M ON M.dep = E.d) E
         ON M.dep = E.d
         AND M.ts=E.ts AND M.te=E.te) X2
    USING(mgr)
    ) Y
GROUP BY mgr, ts, ts;
```

Figure 4.1: Temporal Queries using Temporal Primitives with (l.) and without (r.) WITH.

4.2 The Mapping of Queries with Statement Modifiers to Queries with Temporal Primitives over Algebraic Expressions

Till now, we have discussed the reduction rules and the temporal primitives. We know the relational algebra representations as well as the PostgreSQL syntax of these temporal primitives. Some supporting operations like timestamp propagation as well as the absorb operator have been proposed to ensure the properties of the sequenced semantics and to allow the scaling of attribute values.

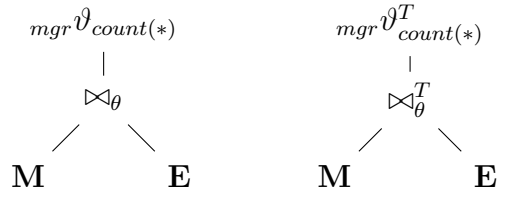
Within the next step we fulfill one of our main goals of this project. We investigate how to achieve the mapping of a query with statement modifiers to a query with temporal primitives. Without statement modifiers and their proper mapping, every time we desire to execute some temporal queries, we would be forced to manually formulate complex queries with temporal primitives. Figure 4.2 makes clear how easier the formulation of temporal queries can be done with temporal statement modifiers. Hiding the complexity of temporal statements will also help to lower the possible error-proneness of expressing intricate temporal queries.

```
WITH X AS (
    SELECT ABSORB mgr, n, M.ts, M.te
    FROM (M ALIGN E ON M.dep = E.d) M
         JOIN
         (E ALIGN M ON M.dep = E.d) E
         ON M.dep = E.d
             AND M.ts=E.ts AND M.te=E.te
)
SELECT mgr, count(*), ts, ts
FROM (X NORMALIZE X USING(mgr)) Y
GROUP BY mgr, ts, ts;
```

```
SEQVT
SELECT mgr, count(*)
FROM M JOIN E ON M.dep = E.d
GROUP BY mgr
```

Figure 4.2: Temporal Queries without (l.) and with (r.) temporal Statement Modifiers.

The mapping started already with the definition of the grammar as described in the previous chapter. The remaining mapping is performed after the grammar rules of the PostreSQL parser create a raw parse tree. This section investigates the mapping of queries over algebra expressions and afterwards in the next section we will determine the mapping in SQL. Figure 4.3(a) shows the query Q in its algebraic representation Q_{ra} as a parse tree, whereas $\theta \equiv (\mathbf{M}.dep = \mathbf{E}.d)$.



(a) Parse Tree (b) Temporal P. T.

Figure 4.3: Mapping Parse Trees

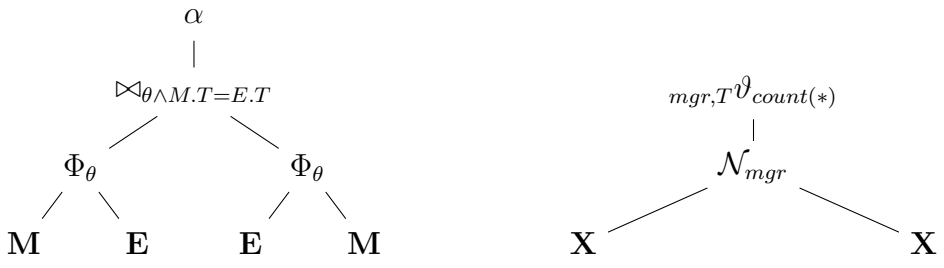
From Chapter 3, we know that the parse tree stands for a select statement which holds in its newly modified version a temporal statement modifier object if the statement modifier keyword `SEQVT` has been set. If so, the parse tree should be processed regarding to the sequenced semantics. To label our algebraic parse tree accordingly, we can mark it by replacing the relational algebra operators $\bowtie$ and ϑ in the parse tree by their temporal representations $\bowtie^T$ and ϑ^T, which is shown in Figure 4.3(b). As it is possible to see, this parse tree is identical to the query Q^T_{ra}. For the next steps and for a better visualization, we split the temporal parse tree into a tree and a sub-tree, where the temporal join is shown as node $\mathbf{X}$ in Figure 4.4(a) and the temporal aggregation in Figure 4.4(b), having $\mathbf{X}$ as a sub-tree.

(a) Temporal Join (b) Temp. Agg.

Figure 4.4: Splitting up the temporal Parse Tree

We proceed with the mapping by reducing the operators with sequenced semantics to their non-temporal counterparts. Hence the temporal join is processed before the temporal aggregation, we reduce the temporal join first, as shown in Figure 4.5(a). As we know it from the reduction rules from Table 2.1, the argument relations $\mathbf{M}$ and $\mathbf{E}$ must both be aligned. Keep in mind that θ is $\mathbf{M}.dep = \mathbf{E}.d$ and therefore the condition for the non-temporal inner join is $\mathbf{M}.dep = \mathbf{E}.d \wedge \mathbf{M}.T = \mathbf{E}.T$.

The next step is to reduce the temporal aggregation as shown in Figure 4.5(b). The reduction needs to perform the normalization by taking the previously reduced temporal join $\mathbf{X}$ as both input arguments of the normalization operator. The normalization has to be processed with respect to the attribute mgr and the grouping for the aggregation be performed with respect to mgr and T. After that, we have the mapping of our example query with statement modifier to its query representation with temporal primitives, that can be executed using PostgreSQL.



(a) Reduced Temporal Join (b) Reduced Temporal Aggregation

Figure 4.5: Reduction on Temporal Operators

To generalize, we know that an ordinary select statement is processed in the order shown in Figure 4.6(a). If such statements are part of a set operation, the set operation has to be

processed after both argument statements have done so. This makes it possible to determine a general case for the order of reduction rules that have to be performed, which is shown in Figure 4.6(b). First, we have to perform the reduction of all the joins and/or Cartesian products placed in the `FROM`-clause. Second, the `WHERE`-clause takes place, but nothing special needs to be done, since a temporal selection is processed the same as a usual selection. Third, we have to apply the reduction rules on aggregates. Fourth, the `HAVING`-clause has to be handled, but since its behavior does not differ from a general selection, there is no special reduction necessary here as well. Fifth, is to take care of the `SELECT`-clause by applying the reduction rules of a temporal-projection. When we processed our ordinary select statement, the reduction rules for set operations is applied if required by the query expression.

(a) General Processing Order

$$
\begin{array}{lrcl}
\texttt{FROM} & r \times_\theta^T s & = & \alpha((r\Phi_{true}s) \bowtie_{r.T=s.T} (s\Phi_{true}r)) \\
 & r \bowtie_\theta^T s & = & \alpha((r\Phi_\theta s) \bowtie_{\theta \wedge r.T=s.T} (s\Phi_\theta r)) \\
 & r \rtimes_\theta^T s & = & \alpha((r\Phi_\theta s) \rtimes_{\theta \wedge r.T=s.T} (s\Phi_\theta r)) \\
 & r \ltimes_\theta^T s & = & \alpha((r\Phi_\theta s) \ltimes_{\theta \wedge r.T=s.T} (s\Phi_\theta r)) \\
 & r \overline{\ltimes}_\theta^T s & = & \alpha((r\Phi_\theta s) \overline{\ltimes}_{\theta \wedge r.T=s.T} (s\Phi_\theta r)) \\
 & r \triangleright_\theta^T s & = & (r\Phi_\theta s) \triangleright_{\theta \wedge r.T=s.T} (s\Phi_\theta r) \\
 & \downarrow & & \downarrow \\
\texttt{WHERE} & \sigma_\theta^T(r) & = & \sigma_\theta(r) \\
 & \downarrow & & \downarrow \\
\texttt{GROUP} & {}_B\vartheta_F^T(r) & = & {}_{B,T}\vartheta_F(\mathcal{N}_B(r;r)) \\
 & \downarrow & & \downarrow \\
\texttt{HAVING} & \sigma_\theta^T(r) & = & \sigma_\theta(r) \\
 & \downarrow & & \downarrow \\
\texttt{SELECT} & \pi_B^T(r) & = & \pi_{B,T}(\mathcal{N}_B(r;r)) \\
 & \downarrow & & \downarrow \\
\texttt{SET OP} & r -^T s & = & \mathcal{N}_A(r;s) - \mathcal{N}_A(s;r) \\
 & r \cup^T s & = & \mathcal{N}_A(r;s) \cup \mathcal{N}_A(s;r) \\
 & r \cap^T s & = & \mathcal{N}_A(r;s) \cap \mathcal{N}_A(s;r)
\end{array}
$$

(b) General Order of the Temporal Operators / Reduction Rules

Figure 4.6: General Case

4.3 The Mapping of Queries with Statement Modifiers to Queries with Temporal Primitives over SQL Expressions

4.3.1 Preliminaries for the actual Mapping

When it comes to the question, what implementation tasks are necessary to integrate the mapping into the PostgreSQL parser, it seems to be useful to analyze this tasks, group these by the aspect of similarity and finally to narrow them down. Regarding Figure 4.6(b) in the last section about the general order of the reduction rules, we can say that we do not have to modify the parser to make the temporal selection work properly. So that means, however the `WHERE`-clause and the `HAVING`-clause are implemented into the PostgreSQL parser, no modifications have to be done there.

Next, we take a look at the temporal anti join $r \triangleright_\theta^T s$. In PostgreSQL, the non-temporal anti join is treated differently than the other joins. This might be more clear when thinking about how the anti join is used compared to the other joins in PostgreSQL. The difference as shown in Figure 4.7 indicates that the internal processing of an anti join is implemented differently in the parser. Whereas the other joins are represented by join-expression objects in the raw parse tree and then further transformed in the `FROM`-clause, the anti join has no resemblance in these points. Basically, the left argument of the anti join represents the main select statement whereas the right argument of the anti join is a subselect nested in the select statement's `WHERE`-clause. The actual anti join and with that the removal of tuples happens in combination with the keyword `NOT EXIST`. After the parsing and analyzing step PostgreSQL's planner recognizes the `NOT EXIST` clause and performs the conversion of the select statement to a join expression. Because of this differences the anti joins needs to be treated as an exceptional case. Hence some non-trivial challenges occur in trying to handle a temporal anti join in the parser, we decided to leave this exceptional case out of the scope of this project and we will deal with it in future work.

```
SELECT *                              SELECT *
[ FROM R JOIN S ON ...                FROM R
| FROM R LEFT JOIN S ON...            WHERE NOT EXIST ( SELECT *
| FROM R RIGHT JOIN S ON...                              FROM S
| FROM R FULL JOIN S ON... ];                            WHERE ...);
```

Figure 4.7: SQL of usual Joins (l.) vs. Anti Join (r.)

Considering the temporal Cartesian product $r \times_\theta^T s$, we might treat its reduction rule as a special case as well. This is not entirely true since there exist two ways to express a Cartesian product in PostgreSQL. As shown in Figure 4.8, one way is to use the keyword `CROSS JOIN` to explicitly characterize the Cartesian product and the other way is to make a comma separated list of `FROM`-clause items which implies a Cartesian product.

As we take an insight in the source code of the parser, we see that the `CROSS JOIN`

```
SELECT *                          SELECT *
FROM M CROSS JOIN E;              FROM M, E;
```

Figure 4.8: Explicit (l.) vs. implicit (r.) Cartesian Product

variant leads to the creation of a join-expression object without a join condition. So it is highly possible that this variant can be implemented similarly to the other joins. In contrast the variant with the comma separated list of items in the FROM-clause is implemented differently, having no join-expression object created after the parsing step. Because of that we have to treat it as an exceptional case. Even more complex, this special case in itself can be divided into two cases. Even though we have a list of items in the FROM-clause, we also may have a join condition inside the WHERE-clause as shown in Figure 4.9. Such a formulation will be transformed by PostgreSQL as an inner join only in the optimizer. Our Implementation has to take care of that and needs to support the processing of such temporal joins for performance reasons as well. Hence no explicit join-expression object is created by the grammar after parsing the inner join formulation, we have to treat these temporal implicit inner joins as exceptional cases too.

```
SELECT *                          SELECT *
FROM M, E;                        FROM M, E
                                  WHERE M.dep = E.d;
```

Figure 4.9: Cartesian product (l.) vs. implicit Inner Join (r.)

In our next observation, we can find some similarities for the temporal aggregation ${}_B\vartheta^T_F(r)$ and the temporal projection $\pi^T_B(r)$. As shown in Figure 4.10, the SQL statement for aggregation is realized by the items in the GROUP-clause as well as from the aggregation functions that are specified in the SELECT-clause. Similarly the projection is realized in the SELECT-clause, containing the desired attributes to project on. The optional grouping possibility for a projection is likewise to the aggregation achieved with the items in the GROUP-clause. Having this strong resemblance to each other, the temporal aggregation and the temporal projection might be implemented in the same breath.

```
SELECT n, sum(l)                  SELECT n
FROM ELOAN                        FROM ELOAN
GROUP BY n;                       GROUP BY n;
```

Figure 4.10: Aggregation (l.) vs. Projection (r.)

All this considerations lead to a number of implementation-tasks as shown in Figure 4.11. The tasks are arranged in the order in which a select statement is processed. The temporal anti join is not the only temporal operator that we leave out of the scope of this project. We do the same with the implicit cross join, the implicit inner join as well as the set operations. Besides the already existing challenges of this project many more major challenges arise in the process to handle these temporal operations and we have to take care of them in future work.

1. Exceptional case: Determine the number of items in the `FROM`-clause. If there is more than one item, transform the items to one or more join-expressions.
   - $r \times_\theta^T s = \alpha((r\Phi_{true}s) \bowtie_{r.T=s.T} (s\Phi_{true}r))$ (implicit cross-join)
2. Investigate the `FROM`-clause and apply the reduction-rules to join-expressions.
   - $r \times_\theta^T s = \alpha((r\Phi_{true}s) \bowtie_{r.T=s.T} (s\Phi_{true}r))$ (explicit cross-join)
   - $r \bowtie_\theta^T s = \alpha((r\Phi_\theta s) \bowtie_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$ (explicit inner-join)
   - $r \rtimes_\theta^T s = \alpha((r\Phi_\theta s) \rtimes_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
   - $r \bowtie\!\!\!\!\subset_\theta^T s = \alpha((r\Phi_\theta s) \bowtie\!\!\!\!\subset_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
   - $r \rtimes\!\!\!\!\subset_\theta^T s = \alpha((r\Phi_\theta s) \rtimes\!\!\!\!\subset_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$
3. Exceptional case: Inspect the `FROM`-clause as well as the `WHERE`-clause and find out if an implicit inner-join needs to be performed and if so, apply the corresponding reduction rule.
   - $r \bowtie_\theta^T s = \alpha((r\Phi_\theta s) \bowtie_{\theta \wedge r.T=s.T} (s\Phi_\theta r))$ (implicit inner-join)
4. Exceptional case: Investigate the `WHERE`-clause for the keyword `NOT EXIST` and decide whether a temporal antijoin has to be transformed.
   - $r \triangleright_\theta^T s = (r\Phi_\theta s) \triangleright_{\theta \wedge r.T=s.T} (s\Phi_\theta r)$
5. Analyze the `SELECT`-clause and apply if needed the reduction-rules for temporal projection and temporal aggregation.
   - $\pi_B^T(r) = \pi_{B,T}(\mathcal{N}_B(r;r))$
   - ${}_B\vartheta_F^T(r) =_{B,T} \vartheta_F(\mathcal{N}_A(r;r))$
6. If a select statement is flagged as a set operation, apply the appropriate reduction rules.
   - $r -^T s = \mathcal{N}_A(r;s) - \mathcal{N}_A(s;r)$
   - $r \cup^T s = \mathcal{N}_A(r;s) \cup \mathcal{N}_A(s;r)$
   - $r \cap^T s = \mathcal{N}_A(r;s) \cap \mathcal{N}_A(s;r)$

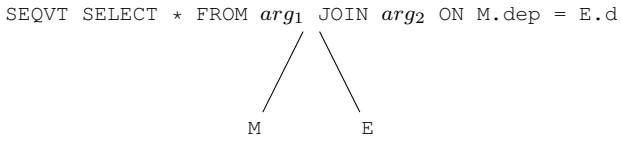
Figure 4.11: Implementation Tasks

4.3.2 Reducing Temporal Join-Expressions

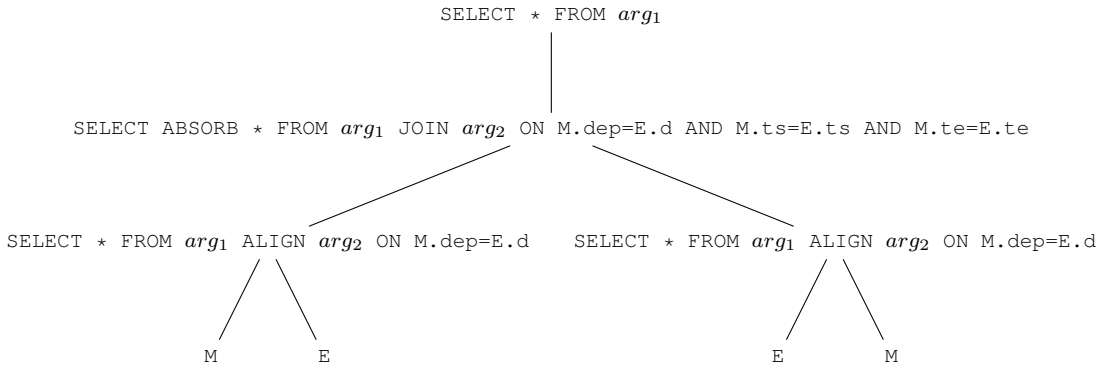
By neglecting a few details we can say that in the transformation process of a select statement the `FROM`-Clause is the first clause that is analyzed in PostgreSQL and then appropriately transformed for the query tree. This corresponds to the SQL processing order discussed in Figure 4.6(a). Basically, the algorithm iterates through all the items in the `FROM`-clause and transforms each of them individually. Each item is tested for its type and each type is transformed in its specific way. There are a few possible types like a plain relation, a subselect or for what we are searching for now, a join-expression. But in the case of our implementation we do not only test the item if it is a join-expression, we also check for the possibility if this item is a join-expression and a temporal statement modifier has been set for the whole select statement. If this test holds true, we transform the join-expression object according to the reduction rules. Having our running example in mind, this transformation is very similar to the mapping shown before in Figure 4.4(a) and Figure 4.5(a). In our implementation we took advantage of the already existing implementations from Dignös et al. [1] and reused their code whenever possible.

Example 9 *Consider the temporal join of the query Q^T or rather as expressed by the algebraic query $Q^T_{ra_{red1}}$. The corresponding SQL fromulation of the original select statement is shown in Figure 4.12(a). At the beginning of our translation algorithm we create two new select statements. These statements represent the alignment operations* `M ALIGN E ON M.dep=E.d` *and* `E ALIGN M ON M.dep=E.d`*. The code that is used to create each of these two statements is in principle the same as the already existing code inside the* `ALIGN`*-rule in the gram.y file (src/backend/parser/gram.y[1]). The difference is that we do not get the columns, relations and qualifiers directly from the users formulated SQL statement, but from the original join-expression that we are trying to reduce. So the left and right arguments* `M` *and* `E` *of the inner join as well as its condition* `M.dep = E.d` *are passed appropriately to both of the alignment statements. As next we have to ensure that the left and right arguments of the join-expression are replaced with the newly created alignment statements. We further extend the conditions of the join with the expression* `AND r.ts = s.ts AND r.te = s.te` *to cope with the reduction rules. Finally, we wrap up the join in a new select statement that has set the* `ABSORB` *keyword. The tree of these newly created statements is shown in Figure 4.12(b). One non-trivial problem arises if we have a closer look at this statement tree. The transformed inner join expression joins the two aligned relations* `M` *and* `E` *together. Since both of the relations possess the interval timestamp attributes* ts *and* te*, we will have unnecessary duplicate timestamps in our result relation. Even worse is the fact that the result would have ambiguous column names. If no further operations follow which are using these timestamps, it would not cause a problem. But keeping in mind that some more temporal operations may follow and as it is in our running example, we need to remove the duplicate timestamps. In Chapter 5 we will investigate some aspects of the implementation of the query transformation very closely and the solution to the duplicate timestamp problem is a part of it. The problem is not trivial as one might think and it will give us some important insights about the implementation of the PostgreSQL parser itself.*

[1] http://www.ifi.uzh.ch/dbtg/research/align.html



(a) Original Temporal Join Statement



(b) Reduced Temporal Join

Figure 4.12: Reduction of Temporal Joins in SQL

This transformation algorithm is valid for all the other join-types (except the anti join) since none of our transformation steps is specific to a type of join-expression to be reduced. Having the alignment statements created the same as in the `ALIGN` grammar-rule, allows us also to let the alignment statements be transformed by the already implemented function `transformAdjustmentStatement` (src/backend/parser/analyze.c[1]) which has also been made available by Dignös et al. [1]. The transformation in this function is responsible for compiling an appropriate SQL expression which does the actual temporal alignment as described by Dignös et al. [1].

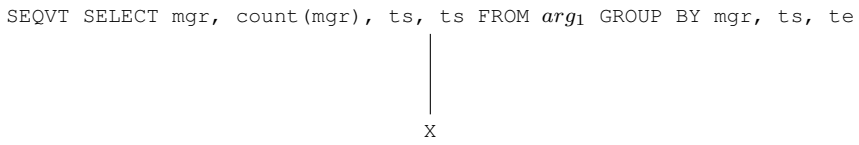
4.3.3 Reducing Temporal Projection and Temporal Aggregation

Since temporal aggregation and temporal projection need to be performed after processing the temporal joins, we have to find the right place to apply the appropriate reduction rules. In the PostgreSQL parser, the transformation order of the different clauses for a select statement is not the same as the final processing order of a select statement as shown in Figure 4.6(a). The `SELECT`-clause is analyzed and transformed to the needs of the query tree right after the transformation of the `FROM`-clause. Only after that, the `WHERE`-, `GROUP`- and `HAVING`-clauses can be transformed, due to their dependency on what stands in the select statement's `SELECT`-clause. This mixed up order of transforming the different parts of a select statement makes it difficult to achieve the correct final order of processing temporal operations. The

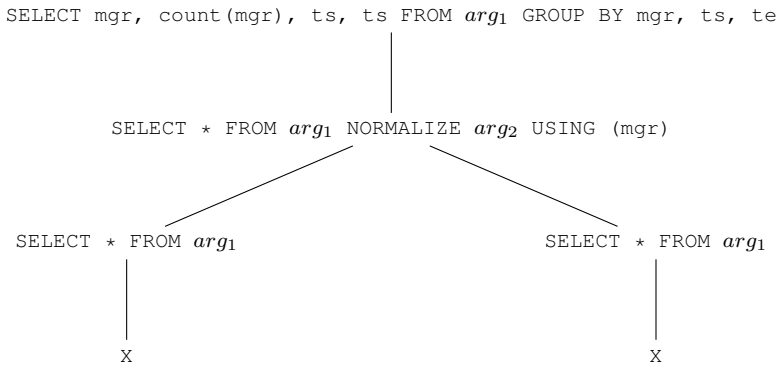
[1] http://www.ifi.uzh.ch/dbtg/research/align.html

reduction of temporal aggregations and projections can not happen in the usual transformation function of a select statement. Instead we have to separate the projection and/or the aggregation from the original statement and put it on top of the transformation of the original select statement. In Example 10 we show the details how the transformation is done.

Example 10 *We show the temporal aggregation of the query Q^T which is also expressed by the algebraic query $Q^T_{ra_{red2}}$. The original SQL statement for the temporal aggregation is shown in Figure 4.13(a). In our implementation of the translation algorithm we first create a new select statement that copies the* `SELECT`*-clause and* `GROUP`*-clause from the original aggregation statement. The new statement wraps up in its* `FROM`*-clause a new adjustment statement containing the normalization operation* `X NORMALIZE X USING(mgr)`*. As before, the code used to create this adjustment statement is nearly the same as the existing code inside the* `NORMALIZE`*-rule which can be found in the parsers grammar file. As it has been the same with the temporal alignments before, we do not get the arguments for the normalization statement from the users input directly. Instead we derive the normalizations* `USING`*-clause from the original select statements* `GROUP`*-clause and extract the timestamp column names. The left and the right arguments for the* `NORMALIZE` *operation are then assigned with our original aggregation statement. At the end we have to ensure that the* `SELECT`*-clause from the original statement is a plain $\star$ so that no projection happens before the temporal normalization operation takes place. Furthermore the* `GROUP`*-clause is removed since the grouping happens inside the wrapper statement which we have created in the beginning of the transformation algorithm. In Figure 4.12(b) we show the newly created tree of statements.*



(a) Original Temporal Aggregation Statement



(b) Reduced Temporal Aggregation

Figure 4.13: Reduction on Temporal Aggregation and Projection in SQL

The transformation algorithm handles temporal aggregations as well as temporal projections since none of the reduction steps has to distinguish between these two different oper-

ations. Similar to the last subsection, having the normalization statement created the same way as in the `NORMALIZE` grammar-rule enables us to let this adjustment statement to be transformed by the `transformAdjustmentStatement` (src/backend/parser/analyze.c[1]) function too. As denoted in the previous subsection, this function has been provided by Dignös et al. [1] and is responsible for transforming the normalization statement to an SQL statement that is capable to perform a temporal normalization.

[1] http://www.ifi.uzh.ch/dbtg/research/align.html

5 Implementation

In Section 4.3 we have proposed the basic approach of our implementation of the mapping of statements with temporal statement modifiers to statements with temporal primitives. This chapter will give some detailed insights of the implementation and the challenges that had to be mastered. The first section gives a brief introduction to the implementation of the PostgreSQL parser itself and the second section shows implementation-details about the temporal statement modifier. The third section investigates the reduction of temporal join-expressions and how the duplicate interval timestamps after join operations are removed. The fourth section shows how our implementation deals with the explicit temporal Cartesian products. In the fifth section we show the algorithm that detects if a select statement contains a temporal aggregation or projection and therefore the appropriate reduction needs to be done.

5.1 Preliminaries

In order to better understand some aspects of our implementation, we establish a certain level of comprehension of the PostgreSQL parser code. The basic data types in the parser architecture are the structures `Node` and `List`. A `List` can contain an unlimited number of objects of the type `ListCell` and each cell points to a `Node`. For simplicity we can say that each structure in the PostgreSQL parser inherits from `Node`. One important `Node` data type is the `SelectStmt` type and as the name denotes it represents a usual SQL select statement. A `SelectStmt` has attributes and the most important ones for our implementation are shown in Figure 5.1. The attribute `targetList` is a pointer to a list which contains the attributes that should be projected by the `SelectStmt`. The other three attributes should be self-explanatory by their names.

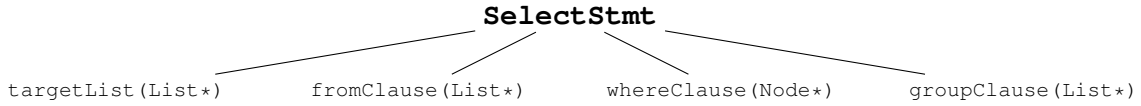


Figure 5.1: The `SelectStmt` Data Type

As seen in Chapter 3, the `simple_select` rule of the PostgreSQL grammar creates such a `SelectStmt` object. This object can also be seen as a raw parse tree and is later further transformed to a query tree, mainly by the `transformSelectStmt` function.

5.2 Determining the Scope of the Statement Modifier

In Chapter 3 we defined that we want to assign a temporal statement modifier object to a select statement that has a prepended `SEQVT`. To make this possible we created a new data type `TempStmtModifier` and added a new attribute `statementModifier` to the `SelectStmt` data type. The `modifiers`-grammar creates such a `TempStmtModifier`-object which is then assigned to the attribute `statementModifier` of the `SelectStmt`. With this we can identify select statements on which we have to apply the reduction rules. If we detect a statement which has a temporal statement modifier, we call a new function `transformTempStmtModifiers()` before the traditional `transformSelectStmt()` is processed. This new function marks the original statement that possible join expressions need to be reduced as shown in the Section 4.3.2. It checks whether a temporal projection or aggregation needs to be done and a normalization tree as shown in the Section 4.3.3 needs be wrapped around the original statement. With the creation of these trees of select statements a new challenge arises. Some of this statements like the alignment, normalization or the reduced join statements need to know the interval timestamps that are currently stored in the statement modifier that is only assigned to the original statement. That leads to the consideration that we need to assign the statement modifier object to these statements as well, so that they have access to the names of the interval timestamps. That in turn leads to the problem that these select statements would be recursively processed by the `transformTempStmtModifiers()` function, that would be wrong since the statement has already been reduced to non-temporal operators. In order to omit this false behavior we add the boolean `isTsmSet` to the statement modifier data type which is shown with its interval timestamp names in Figure 5.2.

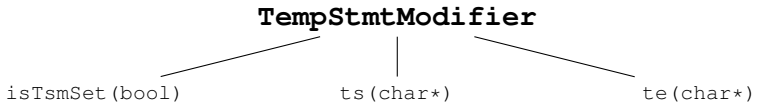


Figure 5.2: The `TempStmtModifier` Data Type

The only time this boolean is set to true is when the PostgreSQL grammar recognizes the keyword `SEQVT` and the original select statement gets the original statement modifier object assigned to it. Now we check select statements not only if they have an assigned `TempStmtModifier` object but also if `isTsmSet` is true. As soon as the statement gets processed by the `transformTempStmtModifiers()` function we ensure that the boolean `isTsmSet` gets set to false. With that, a second invocation of this function is avoided for the rest of the tree of statements while the corresponding statements have access to the statement modifier and the interval timestamp names.

5.3 The Reduction of Temporal Joins

Join expressions are represented in PostgreSQL with the data type `JoinExpr` as shown in Figure 5.3. The `JoinExpr` structure has also some other attributes but we will only briefly introduce the four attributes we need to know for our implementation. The attribute `jointype` stores the type of the join. Some of its possible values are `JOIN_INNER`, `JOIN_LEFT`, `JOIN_RIGHT` or `JOIN_FULL` which indicate respectively the expressions inner join $\bowtie$, left outer join $⟕$, right outer join $⟖$ and full outer join $⟗$. The left and the right argument of a join operation are stored in the attributes `larg` and `rarg` and the `quals` attribute holds the conditions for a join-expression.

Figure 5.3: The `JoinExpr` Data Type

5.3.1 Preventing Endless Reductions on Join-Expressions

In Section 4.3.2 we defined that every `JoinExpr` from a select statement that has a `TempStmtModifier` object will be reduced according to the reduction rules for joins. As previously described in Section 5.2 several statements of the statement-tree which results from the reduction have a `TempStmtModifier` object assigned to them. That leads to the problem that endless recursions of reductions might occur when one of these newly produced select statements hold a join-expression in their `FROM`-clause. Unfortunately, the alignment statements have a left outer join as well as an assigned `TempStmtModifier` so this case would really occur. Now we have to find a mechanism to prevent this from happening and to do the proper implementation. In Section 5.2 we have already denoted a solution to this problem. We said that the `transformTempStmtModifiers()` function marks the original statement that possible join expressions need to be reduced. This is made possible by defining a new attribute `applyJoinReductionRules` for the `SelectStmt` data type which is then set to true by the `transformTempStmtModifiers()` function. We will use this attribute as an additional predicate to check if a join expression needs to be reduced. In Figure 5.4 we show a very simplified version from the function `transformFromClauseItem()` that does the transformation of `FROM`-clause items. This function gets an item node n as input parameter. By using the method $IsA()$ we can determine the type of the node n and transform the node accordingly. By checking the node for its type, verifying the existence of a $TempStmtModifier$ and testing the boolean $applyJoinReductionRules$ we can make sure if the transformation of a temporal join is needed. Since with the `applyJoinReductionRules`-attribute only the original statement may be reduced, an endless reduction is avoided.

Function: `transformFromClauseItem()`

Input: `FROM`-clause item Node n and some other parameters
Output: Tranformed Node n

if $IsA(n, RangeVar)$ **then**
    Transform and return plain relation;
else if $IsA(n, RangeSubselect$ **then**
    Transform and return nested select statement;
else if $IsA(n, RangeFunction)$ **then**
    Transform and return function call appearing in a `FROM`-clause;
else if $IsA(n, JoinExpr) \wedge TempStatementModifier \wedge applyJoinReductionRules$ **then**
    Transform and return temporal join by applying the reduction rules for join-expressions;
else if $IsA(n, JoinExpr)$ **then**
    Transform and return usual join expression;

Throw an error for unrecognized node type of n;
return $NULL$;

Figure 5.4: Determination of the `FROM`-clause Items Type

5.3.2 Removing Duplicate Interval Timestamps

As described in the previous chapter, one major challenge has been the removal of duplicate timestamps after a temporal join operation. Since in the processing of temporal queries both of the aligned argument relations of a temporal join have interval timestamps, we have them twice after the join.

Example 11 *Consider the temporal inner join from our example, we can express the algebraic query $Q^T_{ra_{red1}}$ as the SQL query:* `SEQVT SELECT * FROM M JOIN E ON M.dep = E.d`*. Executing this temporal inner join in terms of our transformation algorithm proposed in the previous chapter would lead to the result relation* $\mathbf{X}$ *as shown in Figure 5.5.*

$\mathbf{X_{dup}}$

	mgr	*dep*	*ts*	*te*	*n*	*d*	*ts*	*te*
x_1	Xavier	1	2012/01/01	2012/09/01	Amber	1	2012/01/01	2012/09/01
x_2	Xavier	1	2012/07/01	2012/12/01	Chelsea	1	2012/07/01	2012/12/01
x_3	Yvonne	2	2012/01/01	2012/04/01	Billy	2	2012/01/01	2012/04/01
x_4	Zoe	2	2012/04/01	2013/01/01	Billy	2	2012/04/01	2013/01/01
x_5	Zoe	2	2012/09/01	2013/01/01	Amber	2	2012/09/01	2013/01/01

Figure 5.5: Relation $\mathbf{X}$ with duplicate Interval Timestamps

The implementation of the PostgreSQL parser possesses a mechanism to junk not required columns in the result relation. Unfortunately this mechanism only allows us to remove columns from the end of a relation. Since the current execution algorithm of the prototype

of adjustment statements requires that interval timestamps are placed at the end of an argument relation, we can not decide to junk the right-most timestamps, and as we will later see this would not be a general solution for all joins. So the provided junking algorithm is no use to us and we have to do our own implementation instead. We have marked the select statement that has a possible temporal join with `applyJoinReductionRules`. So first we let the `FROM`-clause of this statement be transformed which means we let the reduction for join expressions and the whole temporal join happen. After that, the parser is starting with the transformation of the `targetList` to determine which attributes should be projected for the statements result relation. In the case of our temporal join it holds a star $*$ since we do not want our intermediate reduction statements to remove any attributes before the actual projection defined by our original statements `SELECT`-clause happens. In processing the star $*$ the transformation of the `targetList` ensures that it gets every column of all the relations used in the `FROM`-clause. The main processing of the $*$ happens inside the function `expandRelAttrs()` which is responsible to fetch all columns of a single `FROM`-clause item and to return them so that they finally can be stored in the `targetList` of the query tree. So our basic idea is to filter out duplicate interval timestamps while the `expandRelAttrs()` algorithm fetches columns of a `FROM`-clause item.

Example 12 *In order to illustrate our implementation as good as possible, we reuse the employee relation M and use a new relation D where the names of departments are stored as shown in Figure 5.6(a). Note that in relation D we have no tuples that record a department name for department 2 and that d_1 records the fact that department 1 has the name $Ubisoft$ only the first half of the year. We assume the temporal query: $Q_{dup} = \mathbf{M} \rtimes\!\ltimes_{\theta}^{T} \mathbf{D}$ (temporal full join), where $\theta \equiv (\mathbf{M}.dep = \mathbf{D}.dp)$. The result relation $\mathbf{Z}$ is shown in Figure 5.6(b).*

M

	mgr	*dep*	*ts*	*te*
m_1	Xavier	1	2012/01/01	2013/01/01
m_2	Yvonne	2	2012/01/01	2012/04/01
m_3	Zoe	2	2012/04/01	2013/01/01

D

	name	*dp*	*ts*	*te*
d_1	Ubisoft	1	2012/01/01	2012/07/01
d_2	Valve	3	2012/01/01	2012/07/01
d_3	Wahoo	3	2012/07/01	2013/01/01

(a) Argument Relations $\mathbf{M}$ and $\mathbf{D}$

Z

	mgr	*dep*	*ts*	*te*	*name*	*dp*	*ts*	*te*
z_1	Xavier	1	2012/01/01	2012/07/01	Ubisoft	1	2012/01/01	2012/07/01
z_2	Xavier	1	2012/07/01	2013/01/01	ω	ω	ω	ω
z_3	Yvonne	2	2012/01/01	2012/04/01	ω	ω	ω	ω
z_4	Zoe	2	2012/04/01	2013/01/01	ω	ω	ω	ω
z_5	ω	ω	ω	ω	Valve	3	2012/01/01	2012/07/01
z_6	ω	ω	ω	ω	Wahoo	3	2012/07/01	2013/01/01

(b) Result

Figure 5.6: Partial Duplicate Timestamps.

This example clearly shows that the filtering of duplicate timestamps is more complex than expected. Whereas the tuple z_1 has values in both of time intervals, z_2 to z_4 have only values in the timestamp interval attributes coming from the manager relation M. Null values are

denoted by the symbol ω. The tuples z_5 and z_6 have on the contrary only values in the timestamps coming from relation D. That means that our implementation is not allowed to strictly filter out the interval timestamps of one of the argument relations, instead we have to filter duplicate intervals for each tuple individually depending on ω-values.

The solution to overcome this problem is to use the `COALESCE(val1, ..., valN)` function. `COALESCE` can take any number of input arguments and returns the first value that is not null. If all values are equal to null it returns null. Regarding our duplicate timestamp problem this means that we need to use two `COALESCE` expressions, one for the duplicate start point timestamps and one for the duplicate end point timestamps. Figure 5.7 shows a simplified version of the `expandRelAttrs()` function that we have modified.

Function: `expandRelAttrs()`

Input: Some parameters to identify current `FROM`-clause item
Output: $result$, a list of `targetList`-entries

Fetch columns $columns$ from appropriate `FROM`-clause item;
if $TempStmtModifier$ **then**
  foreach $column$ *of* $columns$ **do**
    if $applyJoinReductionRules \wedge column.name = TempStmtModifier.ts$ **then**
      $tscolumns \leftarrow (lappend(tscolumns, column))$;
    else if $applyJoinReductionRules \wedge column.name = TempStmtModifier.te$ **then**
      $tecolumns \leftarrow (lappend(tecolumns, column))$;
    else
      Create `targetList`-entry from $column$ with name $column.name$;
      Append `targetList`-entry to $result$;
  if $list\_length(tscolumns) > 0$ **then**
    Create `COALESCE`-object with $tscolumns$ as input;
    Transform `COALESCE`-object to `targetList`-entry with name ts;
    Append `targetList`-entry to $result$;
  if $list\_length(tecolumns) > 0$ **then**
    Create `COALESCE`-object with $tecolumns$ as input;
    Transform `COALESCE`-object to `targetList`-entry;
    Append `targetList`-entry to $result$;
else
  Execute the usual algorithm to compile the list of `targetList`-entries;
return $result$;

Figure 5.7: Compiling the `targetList` including `COALESCE`-expressions.

Of course, our implementation is only processed if a temporal statement modifier $TempStmtModifier$ is set and therefore its attributes ts and te are set. The algorithm is also aware whether it is processing the `targetList` of a temporal join statement by using the checking the $applyJoinReductionRules$ variable. First, it iterates through the columns of

the FROM clause item that needs currently being processed. If the $applyJoinReductionRules$ variable is set and the current column name equals to ts, we append this ts-column to the temporary list $tscolumns$. If not, we check if the current column name equals to te, and so have to store the te-column in the temporary list $tecolumns$. If none of the both previous cases hold true, we process the $column$ as a non-temporal attribute and append it in a regular manner to the $result$-list. After iterating through the columns we check for both temporary lists $tscolumns$ and $tecolumns$ if timestamps have been fetched. For both cases a COALESCE-object is created that holds the corresponding timestamp list as input value. The COALESCE-object is then transformed to a targetList-entry and appended to the $result$ list.

Example 13 *If we now run the query from example 12, we will get the desired result relation as shown in Figure 5.8. Hence the COALESCE-expression returns the first value that is not null, the interval timestamps from z_1, z_2, z_3 and z_4 are derived from the timestamps from the manager-relations side. On the contrary, the result tuples $z_5 and z_6$ get their timestamp values from the department-relations side.*

Z

	mgr	*dep*	*name*	*dp*	*ts*	*te*
z_1	Xavier	1	Ubisoft	1	2012/01/01	2012/07/01
z_2	Xavier	1	ω	ω	2012/07/01	2013/01/01
z_3	Yvonne	2	ω	ω	2012/01/01	2012/04/01
z_4	Zoe	2	ω	ω	2012/04/01	2013/01/01
z_5	ω	ω	Valve	3	2012/01/01	2012/07/01
z_6	ω	ω	Wahoo	3	2012/07/01	2013/01/01

Figure 5.8: Removed duplicate Timestamps

5.4 The Reduction of the Explicit Cartesian Product

In Figure 4.11 we distinguished between explicit and implicit Cartesian products. We have an explicit Cartesian product when we formulate SQL queries like SELECT * FROM M CROSS JOIN E that contain the terminal CROSS JOIN. The SQL grammar produces a JoinExpr object in these cases and sets its attribute jointype to INNER_JOIN. That's because PostgreSQL treats a CROSS JOIN as an unqualified inner join. Thanks to this circumstance it is relatively easy to handle explicit Cartesian products in the same algorithm that reduces temporal joins as shown in Figure 5.9. As mentioned in Section 4.3.2 the alignment operations $r\Phi_\theta s$ adopt their theta conditions from the join predicates of the joins. Since a CROSS JOIN is an unqualified inner join the alignment operations would fail. But as we have seen in Table 2.1 inside the reduction rule for a Cartesian product, the theta condition for the alignment operations need to hold the value $true$. So the only thing that we have to do is to assign the boolean constant $true$ to the predicates quals of the JoinExpr. This only happens when the quals are $null$ because with that it is sure that a temporal CROSS JOIN needs to be reduced.

```
...
else if IsA(n, JoinExpr) ∧ TempStatementModifier ∧ applyJoinReductionRules then
    ...
    Store node n as JoinExpr j;
    if !j.quals then
        j.quals ← true;
    ...
...
```

Figure 5.9: Dealing with explicit Cartesian Products

5.5 The Reduction of Projection and Aggregation

The reduction of temporal projections and aggregations happens in the `transformTempStmtModifiers()` function. The decision if the appropriate reduction rules need to be applied is made by the function `isTempProjOrAgg()`. Before we introduce this function we briefly explain three other data types from the PostgreSQL parser. One data type is the `ResTarget` structure which is shown in Figure 5.10(a). It is basically used as a wrapper object for an item in the `targetList` of a select statement. Its attribute `val` contains the actual `targetList` item that need to be computed. The second data type shown in Figure 5.10(b) is the `ColumnRef` structure which represents one possible type of a `targetList` item. It is used to specify a reference to a column or to hold a `A_Star` object. The `A_Star` data type is only used to represent the character $*$ that can be used in a `SELECT`-clause and as shown in Figure 5.10(c) it has no real attributes in the parser yet.

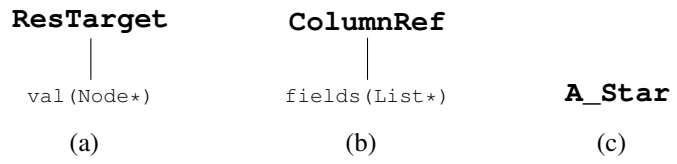


Figure 5.10: The `ResTarget`, `ColumnRef` and `A_Star` Data Types

The basic idea of our implementation is to check whether the `targetList` of the `SelectStmt` is holding only a star $*$. If it is something different from a bare star $*$, we need to do a temporal projection or aggregation. This assumption is a bit naive and not entirely right. There are a few other possibilities how the `targetList` could look like and a temporal projection would not be needed either. For instance when we have a query that has the form `SEQVT SELECT M.*, E.* FROM M, E`. The `targetList` contains the expression $M.*, E.*$, but there is no projection needed since the `targetList` contains the same columns as it is delivered by the relations from the `FROM`-clause. Another sample case that is not correctly covered by our simplified assumption can be shown with the query `SEQVT`

`SELECT mgr, dep, ts, te FROM M`. Again the `targetList` contains all columns that are present in the `FROM`-clause' relation.

Nevertheless we use this assumption for the current prototype, since the un-handled exceptions do not appear that frequently in the common use. The corresponding algorithm of the `isTempProjOrAgg()` function is shown in Figure 5.11.

Function: `isTempProjOrAgg()`

Input: The $targetList$ of a `SelectStmt`
Output: A boolean indicating whether temporal projection is needed

foreach $listCell\ of\ targetList$ **do**
    Assign object pointed by $listCell$ to $restarget$;
    if $IsA(testarget.val, ColumnRef)$ **then**
        Unwrap column reference inside $testarget$ and assign it to $column$;
        if $IsA(llast(column.fields), A\_Star)$ **then**
            if $list\_length(column.fields) == 1$ **then**
                return $false$;
    break;
return $true$;

Figure 5.11: Testing if temporal projection is needed

As denoted we take a `targetList` as input parameter. Each item in a `targetList`-cell is a `ResTarget`-object and holds in its `val` attribute the actual `targetList` item. Common items are of types `ColumnRef` or `FuncCall`. As already introduced, the former represents column names or a star $*$. The latter represents functions, for instance, it may represent aggregate functions. Foremost we fetch the $restarget$ from the first `ListCell` of the `targetList`. We then check if it contains a `ColumnRef` and after that we determine with the condition $IsA(llast(column.fields), A\_Star)$ if the $column$ holds an expression of the form $something.*$. After that we use $list\_length(column.fields) == 1$ to get sure that it is only a bare $*$ expression. If this holds true we return the boolean false and thus avoid that the reduction rules for temporal projection or aggregation is applied. Note that the $ForEach$ loop is only run once and then returns $true$ if no $*$ has been found. Since in our assumption the existence of two or more `targetList` items lead to a temporal projection we do not need any further iterations.

6 Conclusion

In this thesis we describe a mechanism to compile queries with statement modifiers to queries with temporal primitives. As preliminaries for this query compilation, we have looked at reduction rules that are using temporal primitives and thereby define a temporal algebra with sequenced semantics. Then, as part of the query compilation, we have defined an SQL language using temporal statement modifiers for the querying of temporal data. We have illustrated the general case of the SQL mapping by opposing the reduction rules for temporal operations with the processing order of typical SQL queries. The general case of the mapping has been used to define the implementation tasks to finally develop a translation algorithm for the query compilation. In the end we have implemented the translation algorithm in the database management system PostgreSQL. With that we have made the querying of temporal data possible by prepending temporal statement modifiers to a select statement.

Bibliography

[1] A. Dignös, M. H. Böhlen, and J. Gamper. Temporal alignment. In Proceedings of the 2012 international conference on Management of Data, SIGMOD '12, pages 433–444. ACM, 2012.

[2] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. ACM Trans. Database Syst., 25(4):407–456, 2000.

[3] H. Garcia-Molina, J. D. Ullman, and J. Widom. Parsing. In Database systems - the complete book (international edition), chapter 16.1, pages 788-795. Pearson Education, 2002.

[4] M. H. Böhlen, J. Gamper, and C. S. Jensen. An algebraic framework for temporal attribute characteristics. Annals of Mathematics and Artificial Intelligence 46.3, pages 349–374. 2006.

[5] A. Dignös, M. H. Böhlen, and J. Gamper. Query time scaling of attribute values in interval timestamped databases. In Proceedings of the 29th IEEE International Conference Data Engineering, ICDE '13, pages 1304–1307. IEEE, 2013.