

Facharbeit

Implementation of a Relational Algebra Based Graphical User Interface for Temporal Queries

Michael Hartmann

Zürich, Schweiz

Matrikelnummer: 07-711-484

Email: michael.hartmann2@uzh.ch

April 10, 2013

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



University of
Zurich^{UZH}

Department of Informatics



Abstract

In [DBG12] a relational algebra solution, that provides native support for the properties of the sequenced semantics described in [BJS00], has been presented. Temporal expressions are reduced to ordinary expressions using two additional primitive operators to align time intervals. Until now, there is no automatically reduction implemented. In this Facharbeit an application, that let expressions be graphically built and set as sequenced or non-sequenced, is presented. If the expression is set to sequenced it automatically applies the reduction rules and can be executed on a PostgreSQL database system supporting the additional primitives.

Contents

- 1 Introduction 6**
 - 1.1 Reduction from Temporal to Non-Temporal Operators 8
- 2 Problem Definition 12**
 - 2.1 Requirements 12
- 3 Solution 13**
 - 3.1 Application Description 13
 - 3.2 Demonstration of the Example with the Application 15
- 4 Implementation 17**
 - 4.1 Packages 17
 - 4.2 Relations 18
- 5 Conclusions 22**

List of Figures

1.1	Relations <i>proj</i> and <i>mgr</i>	6
1.2	Non-temporal join	7
1.3	Temporal join	7
1.4	Temporal join that scales the budget with respect to the valid time	7
1.5	Normalization of the DB-projects from relation <i>proj</i>	8
1.6	Temporal count of <i>proj</i>	8
1.7	Alignment of relation <i>proj</i> and <i>mgr</i>	9
1.8	Temporal join of <i>mgr</i> and <i>proj</i>	9
1.9	Reduction Rules	10
3.1	Connection Window	13
3.2	Main Window	14
3.3	SQL-Code for creating the relations for the example	15
3.4	Function for uniformly scaling	15
4.1	Structure of the packages	17
4.2	UML class diagram of the relation model classes	18
4.3	Join of two base relations	19
4.4	The <i>asSeqQuery()</i> -method in the class <i>Align</i>	19
4.5	The <i>asSeqQuery()</i> -method in the class <i>Join</i>	20
4.6	Result of calling <i>asQuery()</i> on the sequenced theta join of <i>mgr</i> and <i>proj</i>	21

1 Introduction

In order to process interval timestamped data, the sequenced semantics has been proposed. In [DBG12] a relational algebra solution, that provides native support for the properties of the sequenced semantics, is presented. For this purpose two temporal primitives were introduced and with their support, rules, to reduce temporal operators to non-temporal, have been established. In [BJS00] the notion of universal statement modifiers is introduced. Its main idea is to provide statement modifiers to tag a query as temporal or non-temporal.

The aim of this Facharbeit is to implement a user interface to graphically build relational algebra expressions which can be executed temporal or non-temporal, i.e., sequenced or non-sequenced. Similar to the universal statement modifiers, the user can formulate expressions as traditional and well known relational algebra expressions and is then able to specify which parts of the expression should be treated sequenced or non-sequenced. For each temporal algebra operator the application automatically applies the reduction rules to reduce it to a non-temporal operator that make use of the two primitives.

Example 1 (Temporal Join) Consider projects of a university (Figure 1.1). Projects are stored into the *proj* relation. A project belongs to one department (*d*), has a internal project number (*n*), a budget (*b*) and a start- (*ts*) and ending timestamp (*te*). On the other hand, every department has a manager during a specific time. The managers are stored into the *mgr* relation. In this example for instance Ann managed the database department from 1.5.2013 to 31.7.2013 (m_1) and Sam from 1.8.2013 to 31.12.2013 (m_2). The ordinary non-temporal

proj					
d	n	b	ts	te	
DB	1	181.0	2013-02-01	2013-08-01	p_1
DB	2	196.0	2013-05-01	2014-01-01	p_2
AI	1	153.0	2013-04-01	2013-09-01	p_3
AI	2	120.0	2013-04-01	2013-08-01	p_4

mgr					
r	m	ts	te		
DB	Ann	2013-05-01	2013-08-01	m_1	
DB	Sam	2013-08-01	2014-01-01	m_2	

Figure 1.1: Relations *proj* and *mgr*

join of *mgr* and *proj* is displayed in Figure 1.2. This join doesn't take into account that the timestamps are meant as valid times for the tuples. So for example j_3 gives reason to presume, that Sam was responsible for project number 1, but taking a look on the timestamp, it can be seen that Sam managed the department from 1.8.2013 to 31.12.2013, but in fact project 1 was finished before Sam started (on 31.7.2013).

However, the temporal join of *proj* and *mgr*, i.e., $\text{proj} \bowtie_{r=d}^T \text{mgr}$, takes into account that the tuples hold attributes (*ts*, *te*) for their valid times and includes them into the computation of the join (Figure 1.3). The temporal join aligns the valid times to the time span when both tuples

proj $\bowtie_{r=d}$ mgr								
d	n	b	proj.ts	proj.te	m	mgr.ts	mgr.te	
DB	1	181.0	2013-02-01	2013-08-01	Ann	2013-05-01	2013-08-01	j_1
DB	2	196.0	2013-05-01	2014-01-01	Ann	2013-05-01	2013-08-01	j_2
DB	1	181.0	2013-02-01	2013-08-01	Sam	2013-08-01	2014-01-01	j_3
DB	2	196.0	2013-05-01	2014-01-01	Sam	2013-08-01	2014-01-01	j_4

Figure 1.2: Non-temporal join

are valid, i.e., the intersection of both time intervals. The tuple for Sam mentioned before is not part of the temporal join because the join tuples have disjoint valid times.

proj $\bowtie_{r=d}^T$ mgr						
d	n	b	r	m	ts	te
DB	1	181.0	DB	Ann	2013-05-01	2013-08-01
DB	2	196.0	DB	Ann	2013-05-01	2013-08-01
DB	2	196.0	DB	Sam	2013-08-01	2014-01-01

Figure 1.3: Temporal join

In contrast to the time intervals the other attributes, particularly the budget, are not adapted. In real world scenarios it is frequently requested to scale the attributes with respect to their valid time. An amount of money is a typical example for an attribute which often has to be scaled to represent the reality.

Example 2 (Scaling) In the last example it attracts attention that a sum of the budget (b) over all relations in Figure 1.3 grouped by the project number (n) would distort the amount of budgets. The value of budget for project number 2 would be twice as much as it originally was ($2 \cdot 196.0$), because the whole amount of money is not scaled to the new timestamp that has been adjusted. Figure 1.4 shows the same relation as before, but with scaled attribute budget. The

proj $\bowtie_{r=d}^T$ mgr						
d	n	b	r	m	ts	te
DB	1	92.0	DB	Ann	2013-05-01	2013-08-01
DB	2	73.6	DB	Ann	2013-05-01	2013-08-01
DB	2	122.4	DB	Sam	2013-08-01	2014-01-01

Figure 1.4: Temporal join that scales the budget with respect to the valid time

scaling is uniformly with respect to the duration of the valid time interval. A summation of the budget grouped by the project number would return the original budget ($73.6 + 122.4 = 196$) for project number 2 because the two time intervals meet and cover the original timestamp ($[01.05.2013, 01.8.2013) \cup [01.08.2013, 01.01.2014) = [01.05.2013, 01.01.2014)$). The budget of project 1 differs from the original value, because the valid time of the tuple in the temporal join is only 50.83% of the original one, so the budget is only 50.83% of the original amount also.

Scaling is not a property of the schema, but depends on the purpose of a query. Depending on the semantics of the query an attribute should be scaled or not.

One of the key elements in temporal querying is to adjust the timestamps. In the following section a method from [DBG12], to adjust the timestamps, is presented with the support of two primitives. The primitives align the tuples such that temporal queries then can be transformed to non-temporal ones. Those primitives have been implemented in PostgreSQL and are the basis for this Facharbeit.

1.1 Reduction from Temporal to Non-Temporal Operators

The main problem in temporal operations is that the time intervals of the corresponding tuples can overlap and so an ordinary algebra operator cannot handle it. The idea is to adjust the time intervals of the tuples in order that the operator can use the equality on the time attributes. For this purpose [DBG12] introduced two temporal primitives, temporal normalization \mathcal{N} and temporal alignment ϕ , to adjust the timestamps.

Example 3 (Temporal Aggregation) Consider the relation *proj* of Example 1. Asking for the number of projects in the database department the answer should be: 1 from 1.2.2013 to 30.4.2013 (project 1), 2 from 1.5.2013 to 31.7.2013 (project 1 and 2), and again 1 from 1.8.2013 to 31.12.2013 (project 2).

To process the aggregation by the ordinary (non-temporal) aggregation operator the time intervals of the two argument tuples has to be split, and that is exactly what the temporal normalization operator is designed for. It splits the tuples as shown in Figure 1.5. From the

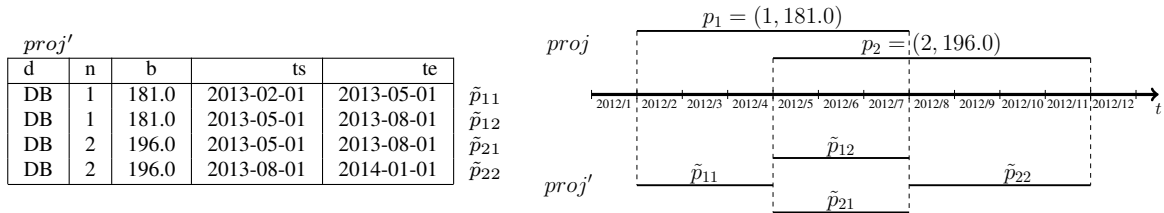


Figure 1.5: Normalization of the DB-projects from relation *proj*

two argument tuples the temporal normalization operator produced four tuples which can be processed by the ordinary aggregation operation (in this case **COUNT**) with the additional grouping arguments *ts* and *te*. So, the SQL code would look like in Figure 1.6.

```
SELECT d, COUNT(*), ts, te
FROM proj'
GROUP BY d, ts, te
```

Figure 1.6: Temporal count of *proj*

The *temporal normalizer* is used for group based operators ($\pi, \vartheta, \cup, -, \cap$). It adjust the time interval of an argument tuple by splitting it at each start and end point of all tuples in the same group.

In contrast, for tuple based operators ($\sigma, \times, \bowtie, \bowtie_d, \bowtie_{\text{d}}, \bowtie_{\text{d}}, \bowtie_{\text{d}}, \bowtie_{\text{d}}$) the *temporal aligner* adjust an argument tuple according to each tuple of a group.

Example 4 (Temporal Join) Consider the relations *proj* and *mgr* from Example 1. To compute the temporal join, $\mathbf{proj} \bowtie_{r=d}^T \mathbf{mgr}$, both the relations have to be aligned with respect to the other relation. Figure 1.7 shows how the temporal aligner adjust the time intervals. For simplicity, only the two tuples p_1 and p_2 are shown, because p_3 and p_4 don't fulfill the θ -condition $d = r$ anyway. The relation *mgr* stays the same, because all the start and end

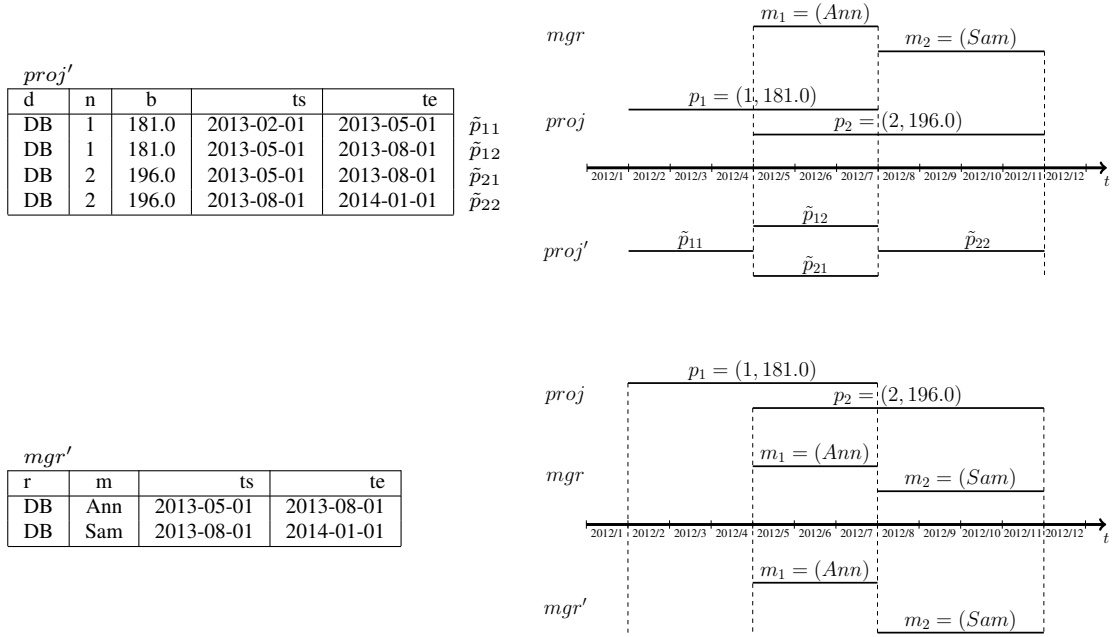


Figure 1.7: Alignment of relation *proj* and *mgr*

points of *mgr* are contained in *proj*. The temporal join can now be computed by applying the ordinary join operator to *mgr'* and *proj'* with additional θ -condition $mgr'.T = proj'.T$, so the corresponding SQL-code is the following:

```

SELECT r, m, d, n, b, ts, te
FROM mgr' INNER JOIN proj'
ON r = d
AND mgr'.ts = proj'.ts
AND mgr'.te = proj'.te

```

Figure 1.8: Temporal join of *mgr* and *proj*

In addition to the two primitives [DBG12] introduced two help operators to define the reduction rules, the timestamp propagation operator ε and the absorb operator α . The *timestamp propagation operator* duplicates the timestamp to be available after adjusting the timestamps. This is for example used for scaling where the scaling function need both, the adjusted and the original timestamp. The *absorb operator* eliminates duplicates that can not be avoided with alignment.

All four operators (the two primitives and the two help operators) are implemented in the extended PostgreSQL-DBMS and are accessible as follows:

$\varepsilon_U(r)$: **SELECT** ts us, te ue, * **FROM** r

$\mathcal{N}_\theta(r, s)$: **FROM** (r NORMALIZE s **USING** θ) r

$\phi_\theta(r, s)$: **FROM** (r ALIGN s **ON** θ) r

$\alpha(r)$: **SELECT** ABSORB * **FROM** r

For every algebra operator [DBG12] defines a rule to reduce the corresponding temporal operator to expression of non-temporal operators. This reduction rules are shown in Figure 1.9.

Operator	Reduction
Selection	$\sigma_\theta^T(\mathbf{r}) = \sigma_\theta(\mathbf{r})$
Projection	$\pi_{\mathbf{B}}^T(\mathbf{r}) = \pi_{\mathbf{B}, T}(\mathcal{N}_{\mathbf{r}, \mathbf{B}=\mathbf{s}, \mathbf{B}}(\mathbf{r}, \mathbf{r}))$
Aggregation	$\mathbf{B} \vartheta_F^T(\mathbf{r}) = \mathbf{B}, T \vartheta_F(\mathcal{N}_{\mathbf{r}, \mathbf{B}=\mathbf{s}, \mathbf{B}}(\mathbf{r}, \mathbf{r}))$
Difference	$\mathbf{r} -^T \mathbf{s} = \mathcal{N}_{\mathbf{r}, \mathbf{A}=\mathbf{s}, \mathbf{A}}(\mathbf{r}, \mathbf{s}) - \mathcal{N}_{\mathbf{r}, \mathbf{A}=\mathbf{s}, \mathbf{A}}(\mathbf{s}, \mathbf{r})$
Union	$\mathbf{r} \cup^T \mathbf{s} = \mathcal{N}_{\mathbf{r}, \mathbf{A}=\mathbf{s}, \mathbf{A}}(\mathbf{r}, \mathbf{s}) \cup \mathcal{N}_{\mathbf{r}, \mathbf{A}=\mathbf{s}, \mathbf{A}}(\mathbf{s}, \mathbf{r})$
Intersection	$\mathbf{r} \cap^T \mathbf{s} = \mathcal{N}_{\mathbf{r}, \mathbf{A}=\mathbf{s}, \mathbf{A}}(\mathbf{r}, \mathbf{s}) \cap \mathcal{N}_{\mathbf{r}, \mathbf{A}=\mathbf{s}, \mathbf{A}}(\mathbf{s}, \mathbf{r})$
Cartesian Product	$\mathbf{r} \times^T \mathbf{s} = \alpha((\phi_\top(\mathbf{r}, \mathbf{s})) \bowtie_{\mathbf{r}, T=\mathbf{s}, T} (\phi_\top(\mathbf{s}, \mathbf{r})))$
Inner Join	$\mathbf{r} \bowtie_\theta^T \mathbf{s} = \alpha((\phi_\theta(\mathbf{r}, \mathbf{s})) \bowtie_{\theta \wedge \mathbf{r}, T=\mathbf{s}, T} (\phi_\theta(\mathbf{s}, \mathbf{r})))$
Left Outer Join	$\mathbf{r} \Join_\theta^T \mathbf{s} = \alpha((\phi_\theta(\mathbf{r}, \mathbf{s})) \Join_{\theta \wedge \mathbf{r}, T=\mathbf{s}, T} (\phi_\theta(\mathbf{s}, \mathbf{r})))$
Right Outer Join	$\mathbf{r} \Join_\theta^T \mathbf{s} = \alpha((\phi_\theta(\mathbf{r}, \mathbf{s})) \Join_{\theta \wedge \mathbf{r}, T=\mathbf{s}, T} (\phi_\theta(\mathbf{s}, \mathbf{r})))$
Full Outer Join	$\mathbf{r} \Join_\theta^T \mathbf{s} = \alpha((\phi_\theta(\mathbf{r}, \mathbf{s})) \Join_{\theta \wedge \mathbf{r}, T=\mathbf{s}, T} (\phi_\theta(\mathbf{s}, \mathbf{r})))$
Anti Join	$\mathbf{r} \triangleright_\theta^T \mathbf{s} = (\phi_\theta(\mathbf{r}, \mathbf{s})) \triangleright_{\theta \wedge \mathbf{r}, T=\mathbf{s}, T} (\phi_\theta(\mathbf{s}, \mathbf{r}))$

Figure 1.9: Reduction Rules

Example 5 (Applying the reduction rule) Consider the same assumptions as in Example 3. The temporal count of projects should be computed, so $d \vartheta_{count(*)}^T(proj)$. The corresponding reduction is given by: $d, T \vartheta_{count(*)}(\mathcal{N}_{proj, \mathbf{B}=proj, \mathbf{B}}(proj, proj))$ which can be stated as follows in SQL:

```

SELECT d, COUNT(*), ts, te FROM
(
  (SELECT * FROM proj) AS proj
  NORMALIZE
  (SELECT * FROM proj) AS proj
  USING(d)
) AS proj
GROUP BY d, ts, te

```

The rest of this Facharbeit is organized as follows: Section 2 defines the requirements for the application, followed by Section 3 which describes how the requirements are solved. The section presents the application and shows how the example from the introduction can be solved with it. Section 4 gives an overview of the implementation.

2 Problem Definition

The two primitives from [DBG12] and the absorb operator have been implemented for PostgreSQL, are accessible as described in the last section, and build the fundament of this Facharbeit. The uniformly scaling, presented in [DBG13], is also implemented as a PL/pgSQL-function.

Until now, there exists no tools that automatically apply the reduction rules to transform temporal to non-temporal relational algebra expressions. The reduction rules has to be applied manually and the expressions has to be written by hand as ordinary SQL expressions extended with the primitives.

To simplify and accelerate the work with a PostgreSQL-DBMS supporting the additional temporal primitives, an application is to be developed that supports the following:

2.1 Requirements

- (R1) Connect to a PostgreSQL-DBMS supporting the additional temporal primitives
- (R2) Queries can be specified as temporal or non-temporal
- (R3) Automatically reduce the temporal (sub-)expressions to non-temporal (sub-)expressions via reduction rules ¹
- (R4) Attributes in temporal queries can be scaled
- (R5) Queries and subqueries can be executed
- (R6) The result set of the last three executed queries should be displayed
- (R7) Display the SQL code sent by the application to the server
- (R8) Display all available tables for the specified database

¹Note that, in the future, the reduction from temporal to non-temporal expressions is thought to be done automatically by the DBMS or some kind of middleware, but this is not implemented for now. So the reduction has to be done by the application itself.

3 Solution

This section first describes the application and then how the example from the introduction can be computed with it.

3.1 Application Description

After starting the application the main screen is in the background and the *Connection Window* is on the top. In this window the user can specify a database connection and connect to it (**R1**). The connection menu can also be reached via the menu bar under *Database > Connection Window*, to change the database connection later. To check the connection there is a button *Ping*, and to confirm the parameters there is a *Connect* button. After connecting the

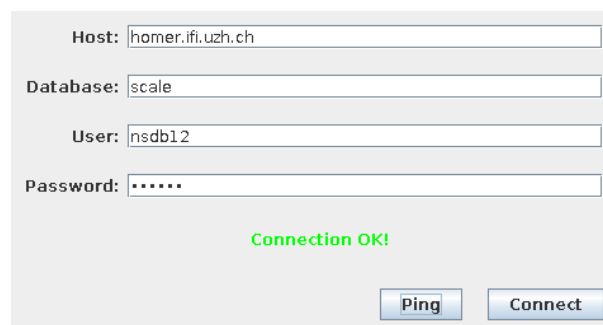


Figure 3.1: Connection Window

Connection Window closes and the *Main Window* appears on the top. The *Main Window* is structured as follows: On the left there is a list of operators that can be used to compose an algebra expression. The symbols are the standard symbols of relational algebra. The symbol called *Base Relation* (\square) selects the whole relation given in the name parameter. The area in the center, called *scene*, is initially empty. By clicking on an operator the symbol for this specific operator appears in this area where it can be freely placed by drag and drop. Operators are selected by just clicking on them.

The right side is split into two parts. In the upper part the parameters (such as sequenced / non-sequenced (**R2**) or scale (**R4**)) for the selected operation are shown and can be modified. The bottom part consists of three tabs, *Relation Set*, *Query*, and *Database Layout*.

- The *Relation Set* displays the history of result sets of the last three executed queries (**R6**). This tab can be detached from the tab plane by rightclicking on it and choosing *Result set in a separate window*. Then the *Relation Set* is a separate window and for example can be placed on a second screen, if available.

- After executing an expression it is possible to see the query in the *Query* tab (R7). This is exactly the same query as the application sent to the server.
- On the *Database Layout* tab there is a list of all available database tables for the specified database connection (R8).

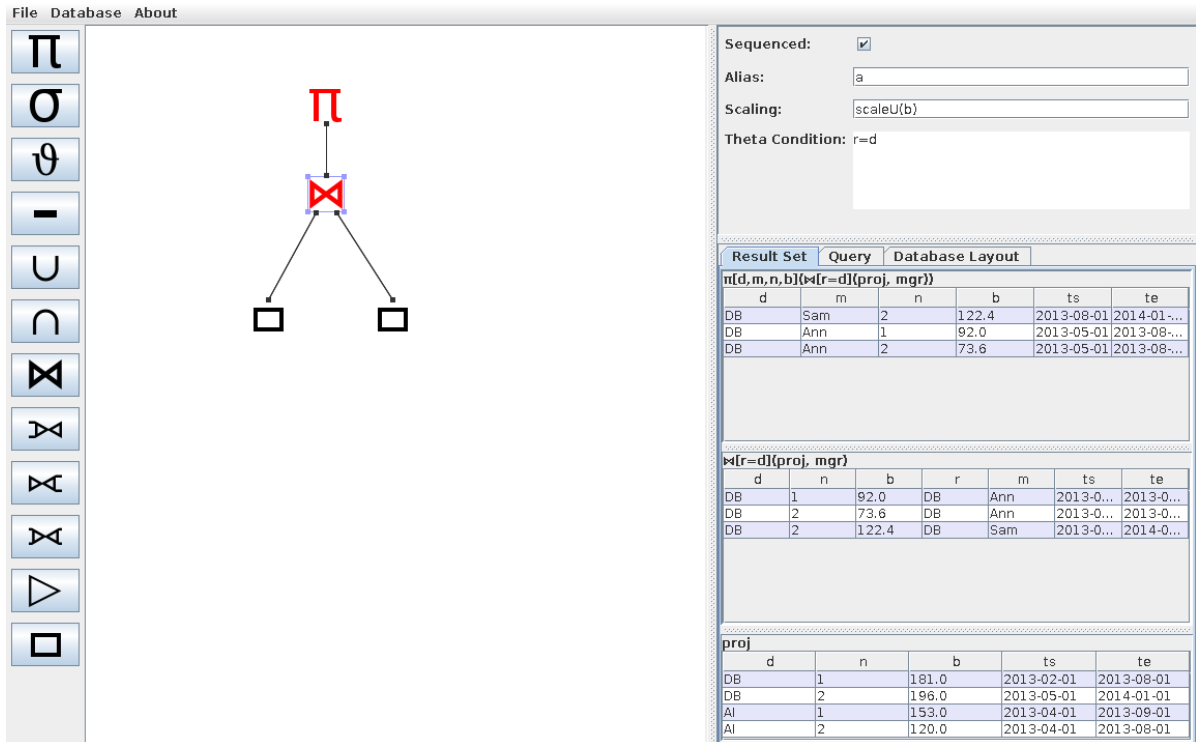
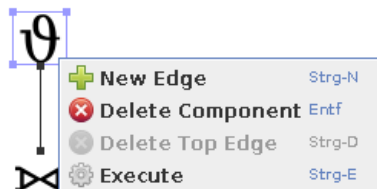


Figure 3.2: Main Window

A popup menu appears by rightclicking on an operator in the scene, and the following four different options are being provided

- *New Edge*: provides a new edge started at the selected operation. By clicking on another component a new edge between these two operation appears.
- *Delete Component*: deletes the selected operation and all edges connected to.
- *Delete Top Edge*: deletes the edge on the top of the operation.
- *Execute*: reduces the temporal part and executes the selected (sub)query (R3), (R5).



3.2 Demonstration of the Example with the Application

This Section demonstrates how the example from the introduction can be computed with the application provided that the relations *proj* and *mgr* exist in the database (see Figure 3.3) and the function for uniformly scaling is defined as shown in Figure 3.4.


```
DROP TABLE IF EXISTS proj;
CREATE TABLE proj (D VARCHAR(2), N INTEGER, B FLOAT, TS DATE, TE DATE);
INSERT INTO proj VALUES ('DB', 1, 181, '2013-2-1', '2013-8-1');
INSERT INTO proj VALUES ('DB', 2, 196, '2013-5-1', '2014-1-1');
INSERT INTO proj VALUES ('AI', 1, 153, '2013-4-1', '2013-9-1');
INSERT INTO proj VALUES ('AI', 2, 120, '2013-4-1', '2013-8-1');


DROP TABLE IF EXISTS mgr;
CREATE TABLE mgr (R VARCHAR(2), M VARCHAR(3), TS DATE, TE DATE);
INSERT INTO mgr VALUES ('DB', 'Ann', '2013-5-1', '2013-8-1');
INSERT INTO mgr VALUES ('DB', 'Sam', '2013-8-1', '2014-1-1');
```

Figure 3.3: SQL-Code for creating the relations for the example

```
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 3.4: Function for uniformly scaling

First step, after connecting to the database, is to place two base relations on the scene for the relations *proj* and *mgr*, by clicking on the base relation icon (). By clicking on one of the appeared icons on the scene the *property panel* for this operation appears on the right top and the property *name* can be filled with the relation name, e.g. 'proj'. Analogous the other base relation can be named 'mgr'. These two base relations can be executed as expressions to detect errors at an early stage. This can be done by right clicking on them and choose *execute* in the popup menu. In general it is always possible to execute subexpressions by clicking on them and choose *execute*.

To join these two relations it requires to add a join operator to the scene. By clicking on the join icon () a new join operator appears on the scene. Every join needs to have a left and a right subquery. This can be done by right clicking on the desired left subquery, say *proj*, and choose *New Edge* and then choose the join operation. Analogous for the right subquery.

By clicking on the join operator the *property panel* appears with four different options (Sequenced, Alias, Scaling, Theta Condition). The join attribute in initial example was the department, so the theta condition should be $d = r$. The alias only matters if the join acts as a subquery, which is not the case in this example, so it can be empty. If the join should be

treated temporal, the checkbox *Sequenced* should be checked and the join operation change automatically its color to red. If the join is temporal then there is also the possibility to scale certain attributes. This can be specified in the *scale* field. In the beginning example the budgeted should be scaled, so in the *scale* field belongs the following: 'scaleU(b)', where 'scaleU' is the function defined in Figure 3.4. Similar to the base relations, the join operator can be executed by right clicking on it and choose execute. The query that was sent by the application can be viewed in the *query tab*. The result set for this executed query is displayed in the *result set tab*. A column of the result set can be ordered ascending or descending by respectively left- or rightclicking on its corresponding header.

4 Implementation

This section gives first an overview of how the packages are structured, then it describes the implementation of the model for the relations and an example to illustrate the commands when executing an expression is given.

4.1 Packages

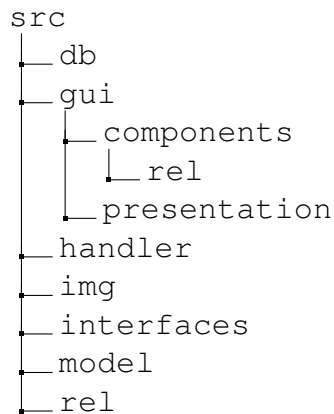


Figure 4.1: Structure of the packages

The structure of the packages is shown in Figure 4.1. Package *db* contains the classes that communicate with a database or treat with data from the database. For instance it contains the class *Executor* and the singleton *ConnectionProvider* for respectively executing queries and providing the connection. The GUI-component are situated in the package *gui*. This package contains the different windows and panels. *gui* is further divided into *components* and *presentation* which contain the components for the scene and the presentation logic, respectively. *components* further contains a subpackage, namely *rel*, that contains the *JComponent* for displaying the relations on the scene and a factory to create them.

The package *handler* contains the class *EventHandler* responsible for updating the relation components on the scene if the underlying model has been changed or vice versa.

The package *img* includes all icons used in the application, to wit the symbols for the operator button as well as the icons for the popup menu.

All interfaces are placed in the package *interface*. The *model* package contains the models for the gui-model-mapping of the relations and the connection settings. Further the *rel* package involves all the models of the relation components. For every relation component which can be placed on the scene there is a corresponding model, compare Section 4.2.

4.2 Relations

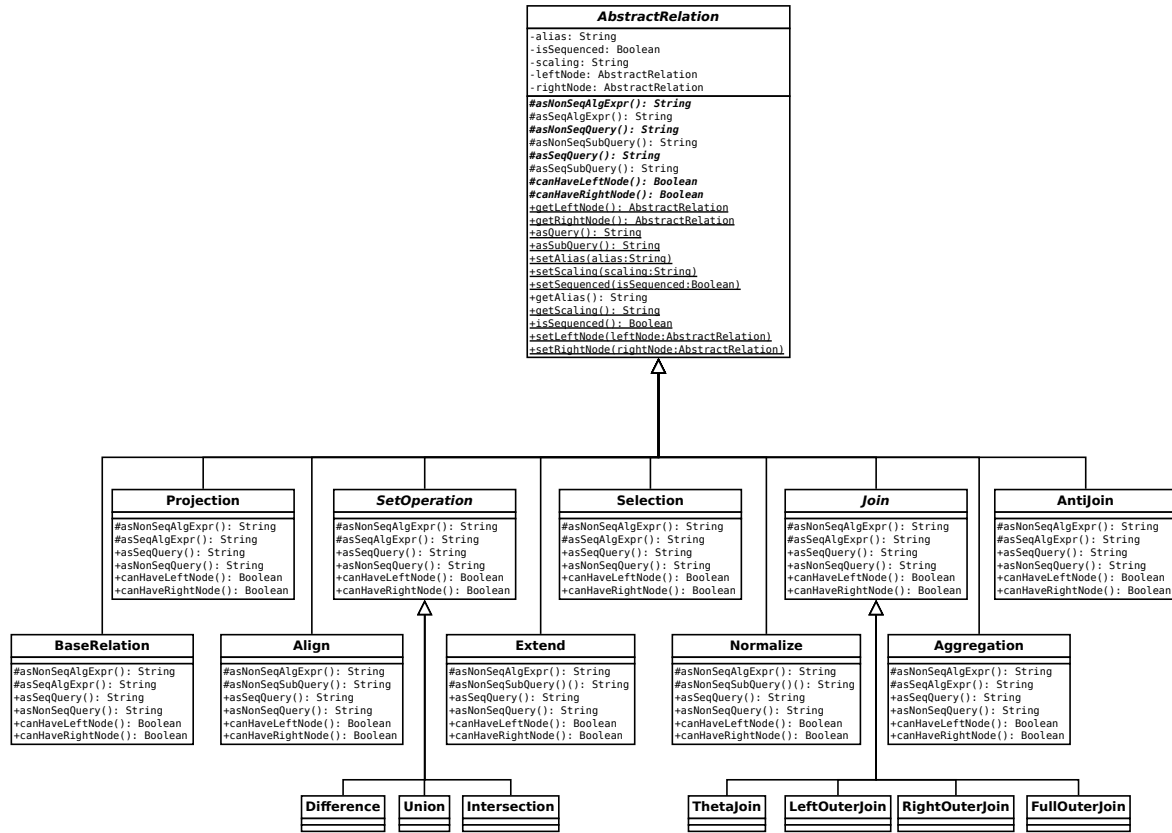


Figure 4.2: UML class diagram of the relation model classes

The relations i.e., either operators or base relation, are located in the package *rel*. Every model of a relation available on the list of operators extends the abstract class *AbstractRelation* directly or indirectly. Figure 4.2 shows this inheritance hierarchy. The additional operators are also extending the *AbstractRelation*. For every relation the user puts on the scene one instance of the corresponding relation model is instantiated. The methods *asNonSeqQuery()*, *asSeqQuery()*, *canHaveLeftNode()*, *canHaveRightNode()* in the *AbstractRelation* class are abstract, so they must be defined by the subclasses.

Example 6 The example illustrates the procedure that is executed when the user clicks on *execute* on a relation on the scene (Figure 4.3). As sample the relations from the last example (*join* of *mgr* and *proj*) are considered. The example is focused on the chain of commands in the model classes and does not go into GUI aspects.

By clicking on *execute* of the temporal join operator the method *asQuery()* of the corresponding instance of a *ThetaJoin* is called. This method, which calls *asSeqQuery()* or *asNonSeqQuery()* depending whether it is sequenced or not, is located in the abstract class *AbstractRelation*. In the example the join relation is sequenced, so *asSeqQuery()* is called. This method is abstract in *AbstractRelation*, so it is implemented by the subclass *Join*. The

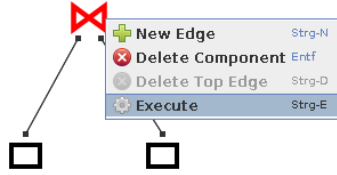


Figure 4.3: Join of two base relations

method is shown in Figure 4.5. First (line 3) the method verifies that a left and a right node is set and throws an exception if this is not the case. If the left and right node are set the method makes a sequence of computations to finally return the query in form of a string. For this, the method first apply propagation operator ε to extend the attributes by copy's of ts and te in the left and right node (line 7-13). In this case the *BaseRelations* are *proj* and *mgr*.

Next a list of the non-temporal attributes are generated for the left and right node (line 15,16), which are used to create the scaling (line 18). In the example the budget will be scaled uniformly. Line 21-29 align the left node with respect to the right and vice versa. In line 31 to 36 the attributes of the left and right node are stored into a list.

The next step (line 38) checks what kind of join it is. In the example, the join is a theta-join, so it jumps directly to line 48, which append the timestamp with prefixed alias of the left node (in the example this is *proj*).

After that the method actually build the query string (line 53-61). For this it makes use of the method *asSubQuery()* from the left and right aligned nodes. This method is shown in Figure 4.4. The method *getTabs()* only adds tab stops to the string to increase the clarity of the resulting SQL code. The *asSubQuery()* of the left and right node (line 3,5 in Figure 4.4) are simple:

(SELECT * FROM baseRelationName) AS baseRelationName

where *baseRelationName* stands for either *proj* or *mgr*.

In total the result of the method *asQuery()* on the *ThetaJoin* is shown in Figure 4.6. This is the query shown in the query tab and which is sent to the server.

```

1  @Override
2  public String asSeqQuery() throws SQLException {
3      String result = getLeftNode().asSubQuery() + "\n";
4      result += getTabs() + "ALIGN\n";
5      result += getRightNode().asSubQuery() + "\n";
6      result += getTabs() + "ON_" + newLines(theta);
7      return result;
8  }

```

Figure 4.4: The *asSeqQuery()*-method in the class *Align*

```

1  @Override
2  protected String asSeqQuery() throws SQLException {
3      if(!hasLeftNode() || !hasRightNode())
4          throw new RuntimeException("ERROR: _ThetaJoin_must_have_two_children!");
5
6      /* extend */
7      Extend lextend = new Extend();
8      lextend.setAlias(getLeftNode().getAlias());
9      lextend.setLeftNode(getLeftNode());
10
11     Extend rextend = new Extend();
12     rextend.setAlias(getRightNode().getAlias());
13     rextend.setRightNode(getRightNode());
14
15     List<Attribute> latts = Attribute.getNonTemporalAttList(lextend.getSchema());
16     List<Attribute> ratts = Attribute.getNonTemporalAttList(extend.getSchema());
17
18     String scaledTheta = Scale.makeScaledCondition(theta, latts, ratts);
19
20     /* align */
21     Align lalign = new Align(scaledTheta);
22     lalign.setAlias(lextend.getAlias());
23     lalign.setLeftNode(lextend);
24     lalign.setRightNode(extend);
25
26     Align ralign = new Align(scaledTheta);
27     ralign.setAlias(extend.getAlias());
28     ralign.setLeftNode(extend);
29     ralign.setRightNode(lextend);
30
31     String tList = Attribute.getAttNameList(latts, null);
32     if(latts.size() > 0)
33         tList += ",_";
34     tList += Attribute.getAttNameList(ratts, null);
35     if(ratts.size() > 0)
36         tList += ",_";
37
38     if(operation == Join.OP_FULL_OUTER_JOIN){
39         tList += "Coalesce("+getLeftNode().getAlias()+". "+Attribute.NAME_TS + ",_"
40             + getRightNode().getAlias()+". "+Attribute.NAME_TS + ")_" + Attribute.NAME_TS
41             + ",_Coalesce("+getLeftNode().getAlias()+". "+Attribute.NAME_TE+" ,_"
42             + getRightNode().getAlias()+". "+Attribute.NAME_TE+" )_" + Attribute.NAME_TE;
43     }
44     else if(operation == Join.OP_RIGHT_OUTER_JOIN){
45         tList += getRightNode().getAlias()+". "+Attribute.NAME_TS+" ,_"
46             + getRightNode().getAlias()+". "+Attribute.NAME_TE;
47     }
48     else{
49         tList += getLeftNode().getAlias()+". "+Attribute.NAME_TS+" ,_"
50             + getLeftNode().getAlias()+". "+Attribute.NAME_TE;
51     }
52
53     String qry = getTabs() + "SELECT_ABSORB_" + tList + "_FROM\n" + lalign.asSubQuery() + "\n"
54     + getTabs() + operation + "\n" + ralign.asSubQuery() + "\n" + getTabs()
55     + "ON_" + newLines(scaledTheta) + "\n" + getTabs() + "AND_" + getLeftNode().getAlias()
56     + ".Ts=" + getRightNode().getAlias() + ".Ts" + "\n" + getTabs()
57     + "AND_" + getLeftNode().getAlias() + ".Te=" + getRightNode().getAlias() + ".Te";
58
59     String scaledTList = Scale.makeScalingTargetList(getScaling(), latts, ratts);
60
61     qry = getTabs() + "SELECT_" + scaledTList + "_FROM\n(" + qry + ")_j";
62
63     return qry;
64 }

```

Figure 4.5: The *asSeqQuery()*-method in the class *Join*

```

SELECT d, n, scaleU(b, Ts, Te, Us, Ue) b, r, m, Ts, Te FROM
(SELECT ABSORB us, ue, d, n, b, vs, ve, r, m, proj.Ts, proj.Te FROM
(
  (
    SELECT Ts Us, Te Ue, * FROM
    (
      SELECT * FROM proj
    ) AS proj
  ) AS proj
ALIGN
  (
    SELECT Ts Vs, Te Ve, * FROM
    (
      SELECT * FROM mgr
    ) AS mgr
  ) AS mgr
ON r=d
) AS proj
INNER JOIN
(
  (
    SELECT Ts Vs, Te Ve, * FROM
    (
      SELECT * FROM mgr
    ) AS mgr
  ) AS mgr
ALIGN
  (
    SELECT Ts Us, Te Ue, * FROM
    (
      SELECT * FROM proj
    ) AS proj
  ) AS proj
ON r=d
) AS mgr
ON r=d
AND proj.Ts=mgr.Ts
AND proj.Te=mgr.Te) j

```

Figure 4.6: Result of calling *asQuery()* on the sequenced theta join of *mgr* and *proj*

5 Conclusions

I have designed and implemented an application, **TheAlignGUI**, that fulfills all the requirements given in Section 2. With this application it is possible to graphically build relational expressions and set them to be executed sequenced or non-sequenced. For the sequenced expressions it automatically applies the reduction rules to be executed on a PostgreSQL database system supporting the additional primitives. It is also possible to set a scaling function which adopts the specified attributes. The query, that is sent to the server, can be seen in the query tab and the result of the last three executed expressions are displayed in the result set tab.

Bibliography

- [BJS00] Michael H. Böhlen, Christian S. Jensen, and Richard Thomas Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, December 2000.
- [DBG12] Anton Dignös, Michael H Böhlen, and Johann Gamper. Temporal alignment. In *ACM SIGMOD 2012 international conference on Management of Data*, SIGMOD '12, pages 433–444. ACM, MAY 2012.
- [DBG13] Anton Dignös, Michael Böhlen, and Johann Gamper. Query time scaling of attribute values in interval timestamped databases. In *29th IEEE International Conference on Data Engineering*, ICDE 2013 (Demonstration), 4 pages. IEEE, 2013.