Freie Universität Bozen

Libera Università di Bolzano

Free University of Bozen · Bolzano

Fakultät für Informatik | Facoltà di Scienze e Tecnologie informatiche | Faculty of Computer Science

Temporal Unification for Database Management Systems

Anton Dignös

supervised by

Prof. Johann Gamper

October 2010

Abstract

Time is present in almost all application domains, and many applications have to store and manage time-varying data. Temporal databases aim to provide specific support for the management of such data. Data have associated one or more time dimensions. The valid time indicates when a stored fact was, is, or will be valid in the modeled reality, whereas the transaction time records when a fact is stored in the database. A lot of research has been conducted in this field over the past decades, focusing mainly on data representation, data models, query languages, indexing, and efficient evaluation algorithms for specific operators. There is little work about integrating temporal support in a DBMS in a principled way. The support for time in commercial database management systems is rather poor, despite the need for the storage and management of temporal data in many applications.

In thesis we provide a novel solution to support time in RDBMS in a principled way. We introduce and define two new operators, termed unary and binary temporal unification, which allow to reduce the temporal operators to the non-temporal counterparts. Temporal unification is a pre-processing step that temporally aligns the argument relations. Then the corresponding non-temporal operators can be applied on the aligned relations. We define reduction rules for the most important operators of a temporal algebra. The reduction to non-temporal operation does not only guarantee snapshot equivalence to the temporal operators, but it preserves also lineage information and allows to take advantage of efficient indexing and evaluation strategies in state of the art database systems. We implemented our solution in the PostgreSQL database management system. The implementation was done in the database system core, by defining an SQL extension for temporal unification and modifying the parser, analyzer, and optimizer accordingly. Two algorithms for unary and binary unification were integrated into the executor unit of PostgreSQL. An extensive empirical evaluation of the PostgreSQL implementation shows the scalability of our solution, and that it clearly outperforms a solution that is based on timestamp normalization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Time is present in all possible application domains, and most of the applications have to capture in one form or another time-varying (or time referenced) data. Examples of such applications can be found in the financial sector, such as accounting and banking, in the medical sector for the management of patient histories, scientific applications, monitoring applications, and in the processing of sensor and streaming data. All these applications have in common that keeping just the current state of the modeled reality is not sufficient, rather they need to keep track of the past and store the history of the relevant data.

Motivated by the need for temporal support in database management systems, considerable research work has been conducted over the last three decades about the management and efficient processing of temporal data. The research ranges from data models and representation and query languages (e.g., [2, 3, 9, 10]) to the development of operators, index structures, and algorithm to efficiently process such data (e.g., [1, 7, 18, 20, 25]).

Despite the need for the storage and management of temporal data and many years of active research on temporal databases, the support for time in most professional and commercially available database management systems is rather limited. The only temporal support offered in databases are a few data types, such as *Date* to store a Calendar date and *Period* to store a time interval in a single attribute. Together with such data types, functions and predicates are provided, e.g., to manipulate dates, to compute the length of an interval, or to compare two intervals. Although, this support makes some expression simpler, e.g., to compute a temporal join we can use the *intersection* function and the *intersection* predicate, it does not help for operations such as union, difference or aggregation. Thus, such extensions do not provide temporal support in a principled way. This situation forces the application programmer to use conventional relational database systems merely as enhanced storage systems and to realize the logic for the processing and management of temporal data in the application program, which makes the program development more complicated, error-prone, and costly.

The limited support for the storage and management of temporal data in (relational) database management systems motivates the research work done in

this thesis. Thus, the main aim of the thesis to provide support for temporal operators in a principled way, using as much as possible from the underlying RDBMS.

1.2 Problem Description

The main problem to process operations such as joins, set operations, and aggregation on temporal data in relational database management systems is due to the mismatch of temporal data and the relational data model. The relational system models the data as tables. A table is vertically organized into rows, also called tuples. The horizontal organization of the table is a fixed set of columns, where each column represents an attribute. The relational model requires the data to be in 1NF (1^{st}-normal form), i.e., each attribute assumes an atomic value.

On the other hand, temporal data is not (or not necessarily) atomic. In this thesis we assume tuple-timestamping, that is each tuple has a special timestamp attribute, T, which records the time interval over which the tuple is valid in the modeled reality. The timestamp is represented as an interval $T = [T_S, T_E)$, where T_S represents the inclusive starting point of the time interval and T_E its non-inclusive ending point. While such a timestamp attribute can clearly be stored in a relation in 1NF (i.e., as an atomic attribute), the semantics of an interval is not just a pair of two numbers, but it represents the *set* of all time points between T_S and T_E. Thus, T is stored as an atomic value, its semantic however is not atomic. This mismatch prohibits to apply the usual set semantics and the corresponding operations to manipulate time intervals. For example, the non-temporal intersection operator will not give the expected result when it is applied to intersect two temporal relations.

Example 1. Consider the small database shown in Figure 1.1, which we will use as a running example throughout the thesis. The database consists of two temporal relations, **r** and **s**. Both relations have the same schema, $(Emp, Dept, T)$, where *Emp* is an employee name, *Dept* is a department, and T is a timestamp. It is obvious that a non-temporal intersection of **r** and **s** would produce an empty result, i.e., $\mathbf{r} \cap \mathbf{s} = \emptyset$, since all tuples in the two relations are different from each other (from a non-temporal perspective). However, the temporal intersection of the two relation should produce two tuples, namely $\mathbf{r} \cap^T \mathbf{s} = \{(\text{Sam}, \text{DB}, [4, 6)), (\text{Joe}, \text{DB}, [14, 19))\}$, since the tuples r_1 and s_1 intersect over the time interval $[4, 6)$, and r_4 and s_2 over the time interval $[14, 19)$.

The major problem for the non-applicability of non-temporal operators for temporal relations is the equality predicate, which is not correctly applied for set-valued attributes, such as the timestamp attribute T. It returns just one value *true* or *false* for the entire timestamp, rather than one value for each time point in the timestamp. For instance, the tuples r_1 and s_1 in Figure 1.1 are obviously not equal if considered as non-temporal tuples. However, with a temporal semantic they are equal over the common sub-interval $[4, 6)$, for which equality predicate should return true.

The usual set semantics of the relational data model can be used when the timestamp attribute represents a single time point. This can be achieved by

r

	Emp	*Dept*	*T*
r_1	Sam	DB	$[1, 6)$
r_2	Ann	DB	$[3, 8)$
r_3	Ann	AI	$[9, 15)$
r_4	Joe	DB	$[14, 19)$

s

	Emp	*Dept*	*T*
s_1	Sam	DB	$[4, 11)$
s_2	Joe	DB	$[12, 21)$

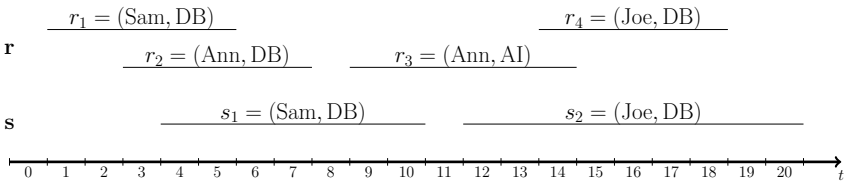


Figure 1.1: Sample Database.

either using a point-based data model [22, 23, 24] or by timestamp normalization [9, 13, 14]. However, such approaches are computationally prohibitive, and they do not preserve lineage information.

Therefore, in this thesis we study the problem of providing support for time in RDBMS in a principled way. Such a solution should be

- efficient,
- preserves lineage information, and
- use current relational database technology as much as possible.

1.3 Contributions

In this thesis we propose a novel solution to support time in RDBMS in a principled way. The core idea of our approach is to split a temporal relation (i.e., its tuples) along the timeline into tuples over maximal time intervals such that all tuples that are valid in such a time interval are constant. In other words, the tuples are aligned along the time dimension. After this alignment of time intervals the equality predicate works correctly. Thus, the set semantics of the relational data model becomes applicable, and non-temporal operators can be used to obtain correct temporal results. Note that this solution preserves lineage information and takes advantages of available database technology, such as indexing and query optimization.

More specifically, the technical contributions of this thesis can be summarized as follows:

- We introduce unary and binary temporal unification as two operators to align temporal relations.
- Using the two temporal unification operators, we reduce the temporal operators to non-temporal operators.
- We implement temporal unification and the reduction of the temporal algebra into non-temporal algebra in PostgreSQL.

- We conduct extensive empirical evaluations of our algorithms, which show the scalability of the proposed solution, which clearly outperforms a solution that is based on normalization.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 discusses related work in the field of temporal databases. Chapter 3 introduces preliminary concepts including the temporal data model used in this thesis. Chapter 4 presents the concept of temporal unification and defines the unary and binary temporal unification operator. Chapter 5 describes the reduction of temporal operators to non-temporal operators using the unification operators. Chapter 6 describes the implementation of the two new operators in the PostgreSQL database system. Chapter 7 evaluates the runtime behaviour of the new operators and compares the performance of our solution with a solution based on timestamp normalization. Chapter 8 concludes and gives directions for future work.

Chapter 2

Related Work

Temporal databases have been an active research since several decades, which investigates various aspects to provide support for the storage and management of time-referenced data in database management systems. In this section we will provide an overview about the most important research directions (including data models, query languages, and query processing techniques) and results achieved so far; we also point to limitations of current technologies, which motivates the work done in this thesis.

2.1 SQL-Based Temporal Query Languages

Dealing with interval based temporal relations using standard SQL is difficult and expensive as illustrated by Richard T. Snodgrass in [19]. In his book, he describes how non-temporal SQL can be used to define and query temporal relations, using operations such as temporal join, different kinds of set operations, etc. Expressing such queries in SQL tends to be extremely large and error-prone, e.g., a simple temporal set difference needs to be decomposed into a UNION of four SELECT statements, each with a nested NOT EXISTS clause, amounting to a total of 47 lines of SQL code. What is worse, such a statement cannot be evaluated efficiently by any current database management system.

The earliest approach to explicitly add time semantics to query languages was to introduce abstract data types in a conventional relational query language, such as SQL. The approach consists in the definition of new data types, predicates, and functions for handling temporal data. The main advantage of this approach is the availability of timestamp data types in the query language and the simplification of operations that involve timestamp attributes. The new predicates are heavily influenced by Allen's 13 interval relations, and they can be applied in selection conditions over time intervals, where otherwise inequalities on the end points are required. Though this approach facilitates the manipulation of single timestamp intervals associated to the data, it does not support the formulation of temporal queries such as aggregation or set difference.

Lorentzos [9, 14] presents the IXSQL language which supports operations on (temporal) intervals. The language uses two operators, *unfold* and *fold*, to normalize timestamps. The unfold operator transforms an interval based relation into a time point based relation, by decomposing each interval timestamped tu-

ple into a set of value-equivalent point timestamped tuples. Then the temporal operations are applied on this time point based intermediate relation. Afterwards, the fold operation transforms the point-based result relation back into an interval representation by collapsing value-equivalent tuples with consecutive time points into maximal intervals. The main drawbacks of this solution are that the intermediate representation does not preserve linage information and its size depends on the time granularity.

A different approach, which is completely based on point timestamps, is SQL/TP proposed by Toman [22]. The main idea is to generalize non-temporal queries to temporal queries. A temporal relation is considered as a sequence of non-temporal relations (or snapshots). On each of the snapshots the non-temporal operations can be applied. While such an approach provides a simple and well-defined semantics, it is unfeasible for any implementation and user interaction.

The TSQL2 query language described in [4, 12] proposes syntactic defaults to make the formulation of temporal queries more convenient. A number of new keywords and clauses are introduced with implicit temporal semantics. While the formulation of temporal queries becomes easier, adding temporal support in a principled and systematic way is difficult with such an approach, since most non-temporal constructs require different and separate extensions.

The problems with TSQL2 are addressed in ATSQL [5], which aims to offer a systematic way to construct temporal queries from non-temporal queries. The main idea of this approach is to first formulate the non-temporal query and then to add a so-called statement modifier which tells the system to evaluate the query in a temporal or non-temporal way.

2.2 Query Processing

A lot of past research is dedicated to the development of efficient temporal query processing strategies, including appropriate indexing structures. The most important operations that have been investigated are temporal join and temporal aggregation.

Temporal joins differ from conventional joins in several ways. First, conventional join techniques are designed for the evaluation of joins with equality predicates. Temporal joins require an intersection predicate, which translate into inequality conditions on the start and end times of the interval timestamps. Second, temporal databases are typically larger than non-temporal databases, since historical data over long time periods are recorded. To efficiently handle such huge amounts of data, specialized techniques are needed. An overview of the most important join evaluation strategies and algorithms is provided in [8]. The work in [8, 27] studies the evaluation of temporal joins with different indexing techniques.

One of the most important and perhaps the most difficult temporal operator is the aggregation, which has been studied in various flavours which mainly differentiate in how the temporal grouping is accomplished. One of the earliest solutions for instant temporal aggregation, where the timeline is divided into time points and for each time point an aggregation group is associated, is the aggregation tree algorithm proposed by Kline and Snodgrass [11]. This work has been improved in [15], where the balanced tree algorithm is proposed, which

avoids worst case scenarios where the aggregation tree ends up in a linear list. Yang and Widom [26] are the first to propose a disk-based index structure for the efficient computation of temporal aggregation in the presence of huge amounts of data such as in data warehouse applications. The temporal multi-dimensional aggregation operator proposed in [1] is a uniform framework which allows to express various forms of temporal aggregation, including instant, moving-window, and span temporal aggregation.

2.3 System Implementations

In spite of the need for temporal support and active research over several decade, commercial database systems provide little support for temporal data.

The integration of temporal support in the *PostgreSQL* database system follows the abstract data type approach. The temporal support is available to the user by extending the database with the temporal module [17]. This module adds the definition of the *Period* datatype, which allows to declare attributes as anchored time-intervals. For the Period datatype, two type of functions are defined, namely boolean predicates and period functions. The former merely allow to evaluate Allen's 13 interval relations between two Period attributes; additionally, comparisons with time points can be done, such as checking whether a time point is contained in an interval. The period functions introduced in the temporal module allow to perform basic calculations on time intervals, e.g., intersection, union, and minus. Since operations on intervals are not closed, these functions might throw a runtime error, e.g., for the union of two disjoint intervals. This module facilitates the formulation of queries over intervals and supports some operations such as the temporal join and intersection, but it does not allow to express queries, where tuples need to be split, such as difference or aggregation.

The *Oracle* database system provide build-in support for all temporal operations that are supported in PostgreSQL. This is the definition of the Period datatype [16] and all predicate and functions associated to it. Additionally, Oracle adds support for valid and transaction time, which is enabled using the DBMS_WM package, which then allows to declare and create temporal relations. Querying temporal relations, however, is only possible at a specific time point; it is not possible, for example, to retrieve the whole history of data, but only to perform queries on single snapshots. Oracle permits either to explicitly specify the time-point in the query or to set an implicit time point for all following queries by using DBMS_WM.GotoDate.

The *Teradata* Database with release 13.10 will become the database system that provides most support for temporal data [21]. Similar to PostgreSQL and Oracle, it will support the Period datatype and all associated functions and predicates. It has valid and transaction time support in order to create temporal tables similar to Oracle, including the capability to perform point queries, i.e., queries on snapshots. In release 13.10 Teradata announced to support also temporal statement modifiers in queries; using the keywords `SEQUENCED` and `VALIDTIME` it is possible to perform temporal updates and temporal queries. However, this support for temporal queries will be limited to simple selects with inner joins. Statement modifiers for outer joins, set operations, duplicate elimination, and aggregation are not supported.

Chapter 3

Preliminaries

In this chapter we introduce some basic concepts and notation about temporal database systems, which are used in the rest of the thesis.

A non-temporal database stores a segment of real world data, which is often refereed to as mini-world. This mini-world describes a set of facts, as for example "The address of client Joe is *Vancouver street* 8" or "Employee Sam is working for department DB". The non-temporal database is kept up to date using *insert*, *update* and *delete*, to adapt the facts described by the database to the current state. By keeping the database up to date modifications to the data are applied, since addresses of clients change and employees might change from one department to another. As soon as a modification is applied, the old data is lost, e.g., it is no longer possible to find the old address of a client.

In order to be able to keep this historical information, time is associated to the data, i.e., the data becomes temporal. By using this time association it is possible to have the whole history of data in one database, and always be able to consult data of the past, although it is not valid now.

In the following the most important properties and concepts of temporal data will be described.

Valid and Transaction Time. The time dimension of temporal data can be of different point of view, i.e., from the storage or from the mini-world point of view. Valid time is a time dimension which references to the mini-world point of view. The information associated to the data is, when was the data *valid* in the mini-world. Such information has to be explicitly specified as the other information to which it is associated. An example of a valid time information is "employee Joe is working for department DB from 1^{st} *May* 2000 till 1^{st} *June* 2000". The time information which is explicitly specified due to the employees contract, gives information about the validity of this fact in the database.

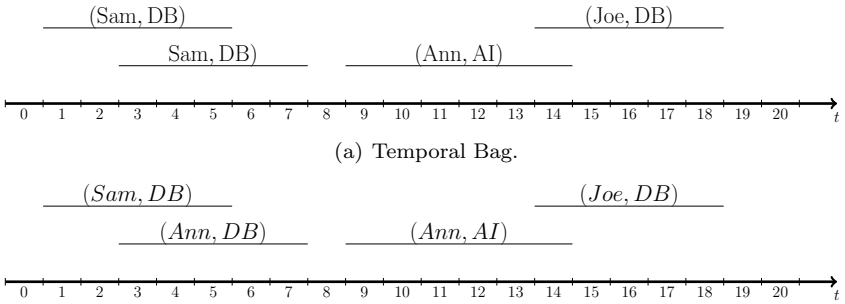
Transaction time is not as valid time explicitly specified, but generated by the database system using the time of the transaction which modified the data. The transaction time associated to the data gives information about, when the data was believed to be valid. An example of transaction time is "employee Joe is working for department DB from 1^{st} *October* 2000 to 2^{nd} *October* 2000". The time in this example does not specify the time the employee Joe was working for department DB, but when it was stored as such. The time in this case

indicates that the information was stored on October 1, but then either removed or modified on October 2.

Time Slices and Snapshots. Temporal databases store the whole history of the data, this means when all tuples from a relation are retrieved then the whole history is returned. To get just the current state of the database, i.e., the state of the mini-world which is valid *now*, the concept of time slices and snapshots have been introduced. A time slice of the data is a cut at a certain point in time, e.g., now, resulting in a snapshot of the data at that time. Therefore a snapshot is a non-temporal relation representing the state of the database at a certain point in time. The snapshot at the time point *now* is equivalent to the state the database would have if it would be non-temporal.

Temporal Sets and Bags. The concept of sets and bags in temporal relations is similar to their conventional definition. A set is a collection of data which has no duplicates, whereas a bag is allowed to have them. The main intuition of a temporal set is, if all possible snapshots of a temporal relation are non-temporal sets, then this relation is a temporal set and it is a temporal bag otherwise. An example of a temporal set and bag will be given later in this section.

Point and Interval Based Relations. In a temporal relation the data has associated timestamps, to represent them although there are various possibilities. The two options which dominate in the temporal databases field are, point based and interval based. The former associates to each data entry exactly one time point, this means in a relation one tuple only represents a fact of a single point in time. To have the same fact on a different time point, a different tuple has to be inserted into the relation. The latter, the interval based temporal model, associates time intervals to the data. This means a single tuple can range over a finite set of consecutive time-points.



(a) Temporal Bag.

(b) Temporal Set.

Figure 3.1: Interval Based Relations.

Figure 3.1 shows a graphical example of two interval based relations, where the time intervals are represented on the time line. For instance we can see the tuple (Joe, DB) ranging over 5 consecutive time points. The same tuple in the point based representation would be a set of 5 different tuples, where each covers just a single time point. In the figure it is also possible to see the difference

between an interval based temporal bag and an interval based temporal set. All snapshots in the temporal set are sets, whereas the snapshot at timepoint 4 in the temporal bag would not result in a set.

Lineage Information. The lineage information is in particular important for interval based relations. The interval stored in a tuple has more information as just the single points would have, i.e., the information about the start and end. Consider the example in Figure 3.1, and the tuples to be contracts of employees, then by looking at the interval we know that the tuple $(\text{Sam}, \text{DB}, [1, 6))$ is a single contract. In the point based representation this information would be either lost or needs to be explicitly specified.

Temporal Data Model. This thesis focuses on interval based temporal sets. The representation of the time intervals is right open $[a, b)$, where a starts the interval and is included, and b ends it but is not included in the interval. When refereeing to intervals the shorthand T is commonly used, which is equivalent to the interval notation $[T_S, T_E)$. Attribute sets are represented by uppercase letters as A or $(A_1, \ldots, A_k)$ is used to refer to a tuple's non-temporal attributes. Relations are denoted by lowercase letters in bold face, commonly $\mathbf{r}$ and $\mathbf{s}$ are used. Tuples of a relation are denoted by lowercase letters in normal style, e.g, $r \in \mathbf{r}$ to denote a tuple r in the relation $\mathbf{r}$.

Chapter 4

Temporal Unification

In this chapter we introduce and define the concept of temporal unification, which is the process of temporally aligning tuples in a temporal relation. This is different from timestamp normalization, which decomposes an interval-timestamped tuple into a set of point-timestamped tuples, thereby loosing any lineage information. Instead, temporal unification transforms an interval-timestamped relation, where tuples are temporally not aligned, into an interval-timestamped relation, where the timestamps are aligned. That is, tuples are decomposed into one or more tuples over smaller yet maximal time intervals such that different tuples either have the same timestamp or can be considered disjoint. Such a unification step preserves lineage information and allows to apply set semantics and the usual equality predicate.

4.1 Unary Temporal Unification

4.1.1 Definition

Unary temporal unification is the process of unifying tuples of a single temporal relation with respect to (equality of) a set of non-temporal attributes, B. That is, a tuple will be unified with all other tuples of the same relation that have identical values for the attributes B. Each tuple is split into one or more tuples over maximal disjoint sub-intervals such that all unified tuples with identical B-values either have equal timestamps or they are disjoint.

Definition 1. (*Unary Temporal Unification*) Let $\mathbf{r}$ be a temporal relation with timestamp attribute T, non-temporal attributes $A = (A_1, \ldots, A_m)$, and $B \subseteq A$. The *unary temporal unification*, $\Phi_B(\mathbf{r})$, of $\mathbf{r}$ with respect to attributes B is defined as follows:

$$
\begin{aligned}
& z \in \Phi_B(\mathbf{r}) \iff \\
& \quad \exists r \in \mathbf{r}(z[A] = r[A] \wedge z.T \subseteq r.T) \wedge \\
& \quad \forall r \in \mathbf{r}(r[B] \neq z[B] \vee z.T \subseteq r.T \vee z.T \cap r.T = \emptyset) \wedge \\
& \quad \forall T \supset z.T \exists r \in \mathbf{r}(r[B] = z[B] \wedge r.T \cap T \neq \emptyset \wedge T \not\subseteq r.T)
\end{aligned}
$$

The first condition requires the existence of a tuple $r \in \mathbf{r}$ from which z takes the non-temporal attribute values and which temporally covers z. The second

condition states that for all tuples, r, that are value-equivalent in the attributes B, either the timestamp $z.T$ is covered by the timestamp $r.T$ or z and r are not overlapping at all. The third condition enforces z to be temporally maximal, i.e., $z.T$ cannot be enlarged without violating condition 2.

Example 2. Consider relation **r** of the running example and consider to unify it with respect to the non-temporal attribute *Dept*. All tuples that match the equality predicate on *Dept* will be unified. The result of unary unification is shown in Figure 4.1. For instance, r_3 and r_4 have different values for the attribute *Dept*, hence no unification is applied, and the two tuples are directly copied to the output. On the other hand, r_1 and r_2 have equal *Dept*-values and they are temporally overlapping. Both tuples are decomposed into two tuples over disjoint time intervals. Two of the new tuples have the same timestamp, namely (Sam, DB, [3, 6)) and (Ann, DB, [3, 6)), whereas the other two new tuples are disjoint.

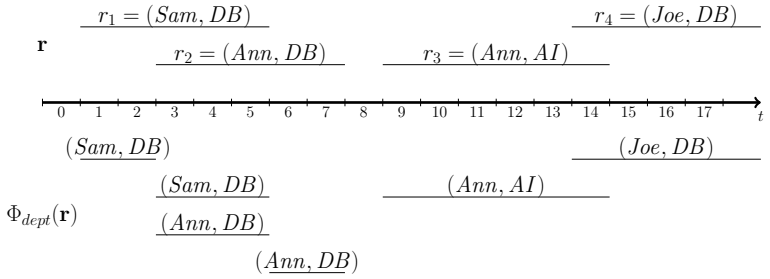


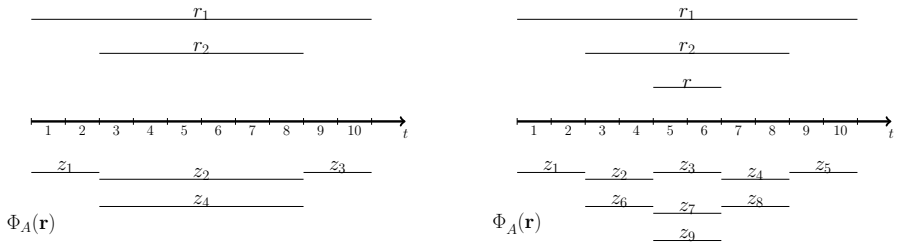
Figure 4.1: Unary Unification.

Theorem 1. *Let* **r** *be a temporal relation with* $|\mathbf{r}| = n$*, and let* $\mathbf{z} = \Phi_B(\mathbf{r})$ *be the result of unary temporal unification with respect to the non-temporal attributes* B*. Then we have* $|\mathbf{z}| \leq n^2$*.*

Proof. We do a proof by induction. Base case: $n = 1$. The result of unifying a relation with one tuple gives one tuple, which is trivially satisfied, since no splits are applied. Inductive case: $n > 1$. Assume that unary unification produces at most n^2 output tuples on an input relation of size n. Then on an input relation of size $n + 1$ at most $(n + 1)^2 = n^2 + 2n + 1$ output tuples are produced. To show that this is correct we argue as follows: n^2 output tuples are produced by n input tuples according to our assumption; one additional input tuple, say r, splits each of the n input tuples into at most three tuples, thus getting $2n$ additional result tuples; and r itself is added to the result. Thus, a single tuple can produce up to $2n + 1$ result tuples. Figure 4.2 illustrates the inductive step from two to three input tuples, where the cardinality of the result increases by $2 * 2 + 1 = 5$ tuples. □

4.1.2 Algorithm

Figure 4.3 shows an algorithm to compute the temporal unary unification. The input parameters are a temporal relation, **r**, with non-temporal attributes A



(a) Two Input Tuples (b) Three Input Tuples

Figure 4.2: Inductive case.

and a set of non-temporal attributes, $B \subseteq A$. The algorithm returns the unified relation $\mathbf{r}$.

The algorithm starts by initializing the result relation $\mathbf{z}$ to the empty set. Then the algorithm iterates over all tuples r of the input relation $\mathbf{r}$ and collects in X all time points, which split r. The first time point is the end point of r itself. Then the algorithm iterates over all tuples $s \in \mathbf{r}$ of the input relation that have the same B-values as r. For each such tuple s the start and the end point of the timestamp are candidates for splitting points. In particular, if such points are covered by the interval $r.T_S$, they are added to X. After processing all tuples s, the variable t_S is initialized to the start time point of r, which will be the start time point of the first sub-interval into which r is split. Then for each time point $t_E \in X$ in chronological order, a new tuple that ends at t_E and has the same non-temporal attributes as r is added to the result relation $\mathbf{z}$. For each new tuple, the end time point t_E will be the start time point of the next tuple.

Algorithm: uUNIFY($\mathbf{r}, B$)

Input: Argument relation $\mathbf{r}$ and set of attributes B.
Output: Result of unary temporal unification $\Phi_B(\mathbf{r})$.

$\mathbf{z} \leftarrow \emptyset$;
foreach $r \in \mathbf{r}$ **do**
  $X \leftarrow \{r.T_E\}$;
  foreach $s \in \mathbf{r}$ *s.t.* $r[B] = s[B]$ **do**
    if $s.T_S > r.T_S \wedge s.T_S < r.T_E$ **then**
      $X \leftarrow X \cup \{s.T_S\}$;
    if $s.T_E > r.T_S \wedge s.T_E < r.T_E$ **then**
      $X \leftarrow X \cup \{s.T_E\}$;
  $t_S \leftarrow r.T_S$;
  foreach $t_E \in X$ *in chronological order* **do**
    $\mathbf{z} \leftarrow \mathbf{z} \cup \{(r.A_1, \ldots, r.A_n, [t_S, t_E))\}$;
    $t_S \leftarrow t_E$;
return $\mathbf{z}$;

Figure 4.3: Unary Unification Algorithm.

Complexity. The complexity of the UNIFY algorithm is $O(|\mathbf{r}|^2)$, where $|\mathbf{r}|$ is the cardinality of the input relation $\mathbf{r}$. For each tuple r of the input relation $\mathbf{r}$ the algorithm iterates over all tuples of $\mathbf{r}$, in order to find its splitting points, these can be at most $2 * |\mathbf{r}| - 1$.

4.2 Binary Temporal Unification

4.2.1 Definition

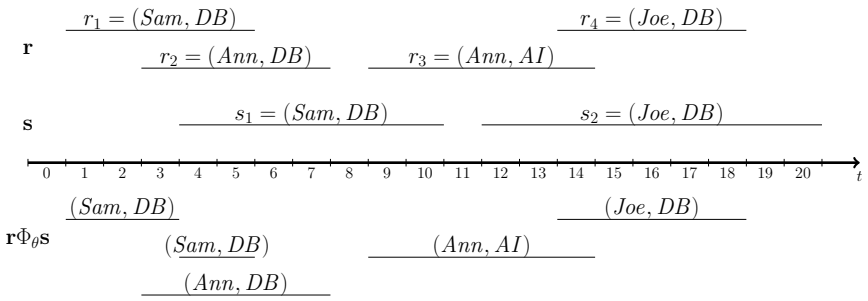
Binary temporal unification is the process of unifying a temporal relation, $\mathbf{r}$, with respect to another temporal relation, $\mathbf{s}$, using a join-condition, θ. A tuple in the argument relation, $\mathbf{r}$, is unified with all tuples in $\mathbf{s}$ for which θ is satisfied. Each tuple $r \in \mathbf{r}$ is split into one or more tuples over not necessarily disjoint sub-intervals of $r.T$ such that there exists a new tuple for each common interval with an $\mathbf{s}$-tuple that matches θ, and the time interval of r is completely covered.

Definition 2. (*Binary Temporal Unification*) Let $\mathbf{r}$ and $\mathbf{s}$ be two temporal relations with timestamp attribute T and let θ be a join-condition over non-temporal attributes between a tuple in $\mathbf{r}$ and a tuple in $\mathbf{s}$. The *binary temporal unification*, $\mathbf{r}\Phi_\theta\mathbf{s}$, of $\mathbf{r}$ with respect to relation $\mathbf{s}$ and condition θ is defined as follows:

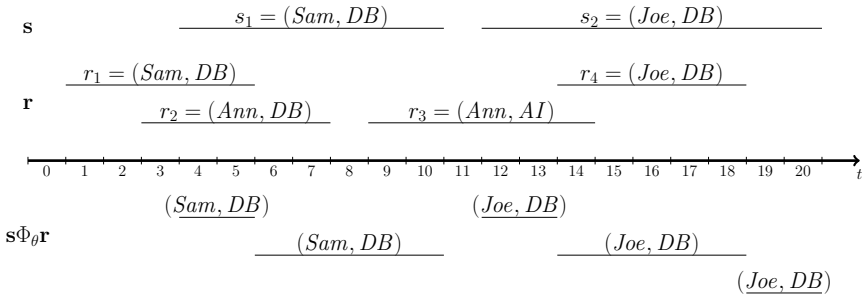
$$
\begin{aligned}
& z \in \mathbf{r}\Phi_\theta\mathbf{s} \Leftrightarrow \\
& \exists r \in \mathbf{r} \exists s \in \mathbf{s}(\theta(r,s) \wedge z[A] = r[A] \wedge \\
& \qquad\qquad z.T = r.T \cap s.T \wedge z.T \neq \emptyset) \vee \qquad (1) \\
& \exists r \in \mathbf{r}(z[A] = r[A] \wedge z.T \subseteq r.T \wedge \\
& \qquad \forall s \in \mathbf{s}(\neg\theta(r,s) \vee s.T \cap z.T = \emptyset) \wedge \qquad (2) \\
& \qquad \forall T \supset z.T \exists s \in \mathbf{s}(\theta(r,s) \wedge s.T \cap T \neq \emptyset \vee T \not\subseteq r.T)
\end{aligned}
$$

The expression of binary unification is a disjunction of two terms 1 and 2. The first term handles the cases where $\mathbf{r}$ and $\mathbf{s}$-tuples satisfy the join condition θ and have a non-empty common sub-interval. For each such common sub-interval a tuple z is in the result relation, which has the same non-temporal attributes as r and the intersection as timestamp attribute, T. The second term handles those sub-intervals of r's timestamp $r.T$, which are not overlapping with any tuple in $\mathbf{s}$ that satisfies θ. For each such sub-interval a tuple z is in the result relation, which has the same non-temporal attributes as r. The last line of expression 2 ensures that the non-overlapping sub-intervals are maximal. It follows directly from the definition that the result tuples specified by the two expressions 1 and 2 are disjoint.

Example 3. Consider relations $\mathbf{r}$ and $\mathbf{s}$ of the running example database and consider to unify $\mathbf{r}$ with respect to $\mathbf{s}$ using the condition $\theta \equiv \mathbf{r}.\mathit{Emp} = \mathbf{s}.\mathit{Emp} \wedge \mathbf{r}.\mathit{Dept} = \mathbf{s}.\mathit{Dept}$. The result of this binary unification operation is shown in Fig. 4.4(a). For instance, the first result tuple, $(\mathit{Sam}, \mathit{DB}, [1,4))$, is produced by r_1 over its sub-interval $[1,4)$, for which no matching tuple in $\mathbf{s}$ exists. The second result tuple, $(\mathit{Sam}, \mathit{DB}, [4,6))$, is produced by r_1 and s_1 over their common time interval $[4,6)$. The binary unification of relation $\mathbf{s}$ with respect to $\mathbf{r}$ using the same θ-condition is illustrated in Fig. 4.4(b).



(a) **r** using **s**



(b) **s** using **r**

Figure 4.4: Binary Unification.

Theorem 2. *Let* $\mathbf{r}$ *be a temporal relation with* $|\mathbf{r}| = n$, $\mathbf{s}$ *be a temporal relation with* $|\mathbf{s}| = m$, *and let* $\mathbf{z} = \mathbf{r}\Phi_\theta\mathbf{s}$ *be the result of binary temporal unification with condition* θ. *Then we have* $|\mathbf{z}| \leq 2nm + n$.

Proof. We do a proof by induction. Base case: $n = 1$. The result of unifying a relation, $\mathbf{r}$, containing one tuple, r, with a relation, $\mathbf{s}$, with m tuples produces at most $2m+1$ result tuples. There are at most m sub-intervals of $r.T$ that overlap with the tuples in $\mathbf{s}$ and at most $m+1$ sub-intervals of $r.T$ non overlapping with any tuple in $\mathbf{s}$. This situation is illustrated in Fig. 4.5, where one tuple in $\mathbf{r}$ produces $2 * 2 + 1 = 5$ result tuples using a reference relation of $m = 2$ tuples. Inductive case: $n > 1$. Assume an argument relation with n tuples can have up to $2nm + n$ output tuples, then $n + 1$ tuples in the input relation can produce $2(n+1)m+(n+1)$ tuples, which is correct, since $2mn+n$ tuples can be produced by n input tuples and an additional tuple can produce up to $2m + 1$ new tuples in the result. □

4.2.2 Algorithm

Algorithm 4.6 shows an algorithm to compute the binary unification. It takes as input an argument relation, $\mathbf{r}$, a reference relation, $\mathbf{s}$, and a join condition, θ, over the non-temporal attributes of $\mathbf{r}$ and $\mathbf{s}$. The algorithm returns the result of binary unification, $\mathbf{r}\Phi_\theta\mathbf{s}$.

The algorithm initializes the result relation $\mathbf{z}$ to the empty set and then starts iterating over all tuples $r \in \mathbf{r}$. For each tuple r it initializes the variable

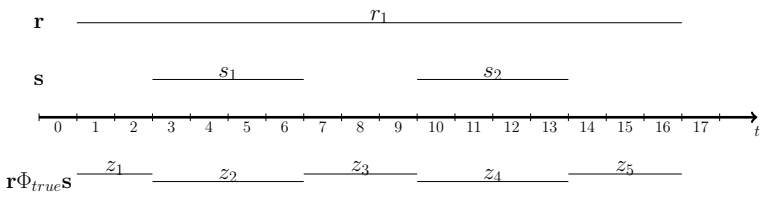


Figure 4.5: Base Case.

t_S to the start time point of r and iterates over all tuples s in the reference relation $\mathbf{s}$ in chronological order, which fulfill the join condition θ and have common time points with r. The variable t_S stores the time point until a result has been produced. First, we check if the tuple r has a sub-interval which is not covered by any s tuple. Since the matching tuples are scanned in chronological order, no succeeding s tuple will cover the interval $[t_S, s.T_S)$. Therefore, it is added to the result and the variable t_S is updated. Next the intersecting part of the tuple r and s is produced and added to the result. The variable t_S is then set to the maximum of its actual value and the end time point of the newly created result tuple. We have to take the maximum, since an already produced tuple could have a higher end time point as the actual one. Once all s tuples are processed, the remaining part of r (if any) not covered by any s tuple is added to the result relation. When all tuples of the argument relation are processed, the algorithm terminates and returns the result relation $\mathbf{z}$.

Algorithm: BUNIFY($\mathbf{r}, \mathbf{s}, \theta$)
Input: Argument relation $\mathbf{r}$, reference relation $\mathbf{s}$ and join condition θ.
Output: Result of binary unification operator $\mathbf{r}\Phi_\theta\mathbf{s}$.

$\mathbf{z} \leftarrow \emptyset$;
foreach $r \in \mathbf{r}$ **do**
  $t_S \leftarrow r.T_S$;
  foreach $s \in \mathbf{s}$ *s.t.* $\theta(r,s) \wedge r.T \cap s.T \neq \emptyset$ *in chr. order* **do**
    if $t_S < s.T_S$ **then**
      $\mathbf{z} \leftarrow \mathbf{z} \cup \{(r.A_1, \ldots, r.A_n, [t_S, s.T_S))\}$;
      $t_S \leftarrow s.T_S$;
    $\mathbf{z} \leftarrow \mathbf{z} \cup \{(r.A_1, \ldots, r.A_n, [\max(r.T_S, s.T_S), \min(r.T_E, s.T_E)))\}$;
    $t_S \leftarrow \max(t_S, s.T_E)$;
  if $t_S < r.T_E$ **then**
    $\mathbf{z} \leftarrow \mathbf{z} \cup \{(r.A_1, \ldots, r.A_n, [t_S, r.T_E))\}$;
return $\mathbf{z}$;

Figure 4.6: Binary Unification Algorithm.

Complexity. The algorithm needs to scan over all tuples of the argument relation $\mathbf{r}$, and for each such tuple all tuples in $\mathbf{s}$ could in the worst case match the join condition θ. In addition the set of matching tuples needs to be sorted resulting in a total complexity of $O(|\mathbf{r}| * |\mathbf{s}| * \log |\mathbf{s}|)$.

Chapter 5

Reduction of Temporal Operators

In this section we show how operators of a temporal algebra can be reduced to non-temporal operators, using the unary and binary unification operators introduce before. For each temporal operator we begin with an informal description, followed by a formal definition, and an illustrative example using our sample database. Then we formulate the reduction rule as a theorem and prove its correctness. An overview of the reduction rules is given in Table 5.1.

Operator		**Reduction**
Projection	$\pi_B^T(\mathbf{r})$	$\pi_B(\Phi_B(\mathbf{r}))$
Aggregation	${}_G\vartheta_F^T(\mathbf{r})$	${}_{G,T}\vartheta_F(\Phi_G(\mathbf{r}))$
Difference	$\mathbf{r} -^T \mathbf{s}$	$(\mathbf{r}\Phi_{\mathbf{r}[A]=\mathbf{s}[A]}\mathbf{s}) - (\mathbf{s}\Phi_{\mathbf{r}[A]=\mathbf{s}[A]}\mathbf{r})$
Intersection	$\mathbf{r} \cap^T \mathbf{s}$	$(\mathbf{r}\Phi_{\mathbf{r}[A]=\mathbf{s}[A]}\mathbf{s}) \cap (\mathbf{s}\Phi_{\mathbf{r}[A]=\mathbf{s}[A]}\mathbf{r})$
Union	$\mathbf{r} \cup^T \mathbf{s}$	$(\mathbf{r}\Phi_{\mathbf{r}[A]=\mathbf{s}[A]}\mathbf{s}) \cup (\mathbf{s}\Phi_{\mathbf{r}[A]=\mathbf{s}[A]}\mathbf{r})$
Cartesian Product	$\mathbf{r} \times^T \mathbf{s}$	$(\mathbf{r}\Phi_{true}\mathbf{s}) \bowtie_{\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_{true}\mathbf{r})$
Join	$\mathbf{r} \bowtie_\theta^T \mathbf{s}$	$(\mathbf{r}\Phi_\theta\mathbf{s}) \bowtie_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$
Left Join	$\mathbf{r} ⟕_\theta^T \mathbf{s}$	$(\mathbf{r}\Phi_\theta\mathbf{s}) ⟕_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$
Right Join	$\mathbf{r} ⟖_\theta^T \mathbf{s}$	$(\mathbf{r}\Phi_\theta\mathbf{s}) ⟖_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$
Full Join	$\mathbf{r} ⟗_\theta^T \mathbf{s}$	$(\mathbf{r}\Phi_\theta\mathbf{s}) ⟗_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$

Table 5.1: Temporal Reduction Rules.

5.1 Projection

The projection operator in temporal databases is a unary operator, which extracts a subset of the non-temporal attributes and the timestamp attribute, T. It has the form $\pi_B^T(\mathbf{r})$, where $\mathbf{r}$ is a temporal relation and B is a subset of the non-temporal attributes of $\mathbf{r}$. The operator outputs a temporal relation which has the schema (B, T).

The projection operator can potentially cause duplicates in the result. That is, although the input is a temporal set, the output might contain duplicates, if candidate keys are removed. To retain sets, the projection operator has to be followed by a duplicate elimination step. Duplicate elimination in temporal databases is often referenced as coalescing [6], which merges value equivalent tuples (in the non-temporal attributes) that are overlapping or adjacent, thus removing the so-called sequenced duplicates [19] of a relation. In this thesis we define duplicate elimination different from coalescing based on the notion of *constant intervals* [1]. The reason for this definition is to keep the correlation of duplicate elimination and aggregation the same as for non-temporal data, i.e., duplicate elimination is equivalent to aggregation with no aggregation function using grouping over all attributes.

Throughout this thesis we assume a duplicate eliminating temporal projection which always produces sets in the output.

Definition 3. The *temporal projection* of a relation $\mathbf{r}$ on attributes $B \subseteq A$ is defined as

$$\begin{aligned}\pi^T_B(\mathbf{r}) = \{z \mid \exists r \in \mathbf{r}(z[B] = r[B] \wedge z.T \subseteq r.T) \wedge \\ \forall r \in \mathbf{r}(r.T \supseteq z.T \vee r.T \cap z.T = \emptyset \vee r[B] \neq z[B]) \wedge \\ \forall T' \supset z.T \exists r \in \mathbf{r}(T' \not\subseteq r.T \wedge T' \cap r.T \neq \emptyset \wedge r[B] = z[B])\}\end{aligned}$$

The first line requires the existence of an r-tuple from which z takes the values of the projected attributes, B, and whose time interval contains $z.T$. The second and third lines require $z.T$ to be a constant interval over all tuples that have the same values for B as z.

Example 4. Figure 5.1 illustrates the temporal set projection on our running example relation $\mathbf{r}$. For instance, the result tuple $(\text{DB}, [3, 6))$ is produced by eliminating the duplicates among the tuples r_1 and r_2.

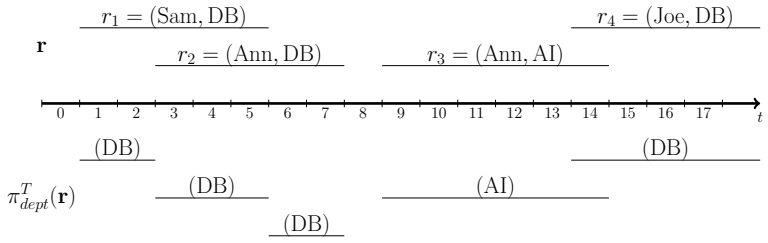


Figure 5.1: Temporal Projection.

The following theorem shows how, by using unary temporal unification, the temporal projection can be reduced to non-temporal projection.

Theorem 3. *Let* $\mathbf{r}$ *be a temporal relation with non-temporal attributes* A *and timestamp attribute* T*, and let* B *be a subset of* A*. The temporal projection on* $\mathbf{r}$ *can be reduced to non-temporal projection as follows:*

$$\pi^T_B(\mathbf{r}) \equiv \pi_{B,T}(\Phi_B(\mathbf{r})),$$

where π *is the duplicate eliminating projection.*

Proof. We show the equivalence of the expression produced by the right-hand side of the theorem and the definition of temporal projection. By expanding $\Phi_B(\mathbf{r})$ in Theorem 3, we obtain an expression that is identical to the definition of temporal projection.

$$\begin{aligned}\pi^T_B(\mathbf{r}) \equiv \{z \mid\ & \exists r \in \mathbf{r}(z'[A] = r[A] \wedge z'.T \subseteq r.T) \wedge \\ & \forall r \in \mathbf{r}(r[B] \neq z'[B] \vee r.T \supseteq z'.T \vee r.T \cap z'.T = \emptyset) \wedge \\ & \forall T \supset z'.T \exists r \in \mathbf{r}(r[B] = z'[B] \wedge r.T \cap T \neq \emptyset \wedge T \not\subseteq r.T) \wedge \\ & z[B] = z'[B] \wedge z.T = z'.T\}\end{aligned}$$

Note that in both cases, in the definition of temporal projection and in the definition of non-temporal duplicate elimination, duplicates are eliminated due to the used set semantics. □

From the definition of unary temporal unification follows that all unified tuples (with respect to attributes B) either have equal timestamp attributes or are disjoint. This implies that all duplicates produced by the projection are equal over all non-temporal and temporal attributes.

Example 5. Figure 5.2 illustrates the computation of the temporal projection on attribute *Dept* using Theorem 3. The first step is to apply unary unification with respect to the *Dept* attribute, which is shown in the upper part. Notice the two tuples $(\text{Sam}, \text{DB}, [3, 6)$ and $(\text{Ann}, \text{DB}, [3, 6)$ with the same *Dept*-value and the same timestamp. Applying non-temporal projection on unification result produces the intended result, which is shown in the lower part.

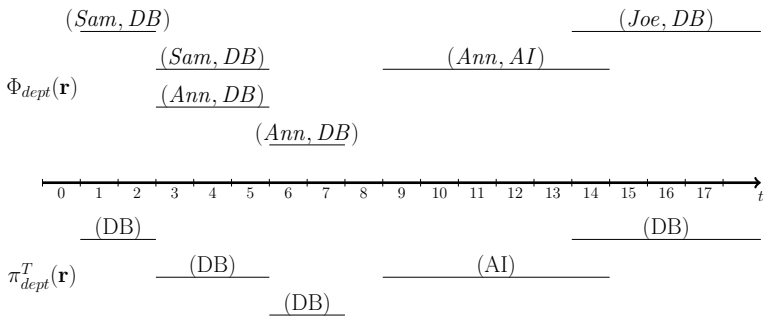


Figure 5.2: Temporal Projection Reduced to Non-Temporal Projection.

5.2 Aggregation

The instant temporal aggregation operator is a unary operator which requires a subset of non-temporal attributes, G, of the input relation and a set of aggregation functions, F, as parameters. For each time point, it evaluates the functions in F over all tuples that are value-equivalent in the attributes G. In other words, the operator produces a non-temporal aggregation for each time-point. Since we use interval-timestamped data and we want to preserve lineage information, the result is coalesced into *constant intervals*.

Definition 4. The *temporal aggregation* of a relation $\mathbf{r}$ using grouping attributes $G \subseteq A$ and aggregation functions F over attributes A is defined as

$$
\begin{aligned}
{}_G\vartheta^T_F(\mathbf{r}) = \{z \mid \exists r \in \mathbf{r}(z[G] = r[G] \wedge z.T \subseteq r.T \wedge z[F] = F(\mathbf{g})) \wedge \\
\mathbf{g} = \{r'[A] \mid r' \in \mathbf{r} \wedge r'[G] = z[G] \wedge r'.T \cap z.T \neq \emptyset\} \wedge \\
\forall r \in \mathbf{r}(r.T \supseteq z.T \vee r.T \cap z.T = \emptyset \vee r[G] \neq z[G]) \wedge \\
\forall T' \supset z.T \exists r \in \mathbf{r}(T' \not\subseteq r.T \wedge T' \cap r.T \neq \emptyset \wedge r[G] = z[G])\}
\end{aligned}
$$

The first condition requires the existence of a tuple $r \in \mathbf{r}$ from which z takes its grouping attributes G, whose interval contains $z.T$, and computes the aggregation functions F over the grouping set $\mathbf{g}$. The second line builds up the grouping set $\mathbf{g}$, i.e., all tuples $r' \in \mathbf{r}$ which have equal attributes for G as z and contain its time interval $z.T$. The last two lines require $z.T$ to be a constant interval over all tuples that have the same values for G as z.

Example 6. Figure 5.3 illustrates temporal aggregation using our running example relation, $\mathbf{r}$, as input relation, the attribute *Dept* for grouping, and the COUNT aggregation function. For instance, the result tuple $(\text{DB}, 1, [3, 6))$ is produced by counting the occurrences of tuples with *Dept* value DB over that interval, i.e., tuples r_1 and r_2.

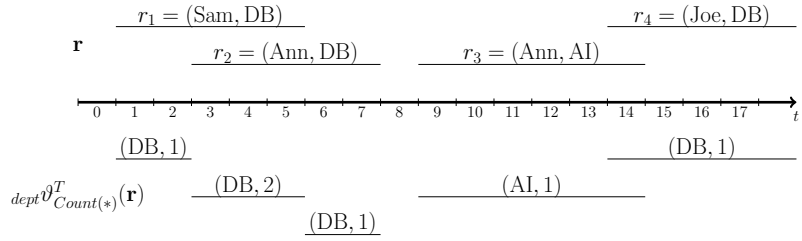


Figure 5.3: Temporal Aggregation.

The following theorem shows how, by using unary temporal unification, the temporal aggregation can be reduced to non-temporal aggregation.

Theorem 4. *Let* $\mathbf{r}$ *be a temporal relation with non-temporal attributes* A *and timestamp attribute* T*. The temporal aggregation on* $\mathbf{r}$ *using grouping attributes* G *and aggregation functions* F *can be reduced to non-temporal aggregation as follows:*

$$
{}_G\vartheta^T_F(\mathbf{r}) \equiv {}_{G,T}\vartheta_F(\Phi_G(\mathbf{r}))
$$

Proof. We show the equivalence of the expression produced by the right-hand side of the theorem and the definition of the temporal aggregation. Thus, we have to show the following:

$$
\begin{aligned}
{}_G\vartheta^T_F(\mathbf{r}) \equiv \{z \mid \exists r' \in \Phi_G(\mathbf{r})(z[G] = r'[G] \wedge z.T = r'.T) \wedge & \quad (1) \\
\mathbf{g} = \{\{r' \mid r' \in \Phi_G(\mathbf{r}) \wedge r'[G] = z[G] \wedge r'.T = z.T\}\} \wedge & \quad (2) \\
z[F] = F(g)\} & \quad (3)
\end{aligned}
$$

Consider the sub-expression 1. By expanding the definition of $\Phi_G(\mathbf{r})$, we get

$$\begin{aligned}
&\exists r \in \mathbf{r}(z'[A] = r[A] \wedge z'.T \subseteq r.T) \wedge \\
&\forall r \in \mathbf{r}(r[G] \neq z'[G] \vee r.T \supseteq z'.T \vee r.T \cap z'.T = \emptyset) \wedge \\
&\forall T \supset z'.T \exists r \in \mathbf{r}(r[G] = z'[G] \wedge r.T \cap T \neq \emptyset \wedge T \not\subseteq r.T) \wedge \\
&z[G] = z'[G] \wedge z.T = z'.T
\end{aligned}$$

The resulting expression is equivalent to the first, third, and fourth line of the definition of temporal aggregation. It remains to show that $\mathbf{g}$ produces in both cases the same relation, for which we briefly sketch the intuition. Consider the sub-expression 2. $z.T$ and $r'.T$ are both bound to a unified interval according to the attributes G. From the definition of unary unification all these intervals have the same timestamps or are disjoint. Therefore, $\mathbf{g}$ in the definition of temporal aggregation and in the theorem are the same relations for the same tuple z, except that in the theorem the tuple r' is not projected according to its non-temporal attributes A. The projection does not cause any conflict, since $\mathbf{g}$ is a bag, therefore no duplicate elimination is applied and the aggregation functions F are not allowed to be computed over timestamp attributes. □

Example 7. Figure 5.4 illustrates the computation of the temporal aggregation using Theorem 4. The first step is to apply the unary unification with respect to the *Dept* attribute, which produces two tuples $(\text{Sam}, \text{DB}, [3, 6)$ and $(\text{Ann}, \text{DB}, [3, 6)$ with the same *Dept*-value and the same timestamp. Then, by applying non-temporal aggregation on the unification result we get the intended result, in particular the value 2 over the interval $[3, 6)$.

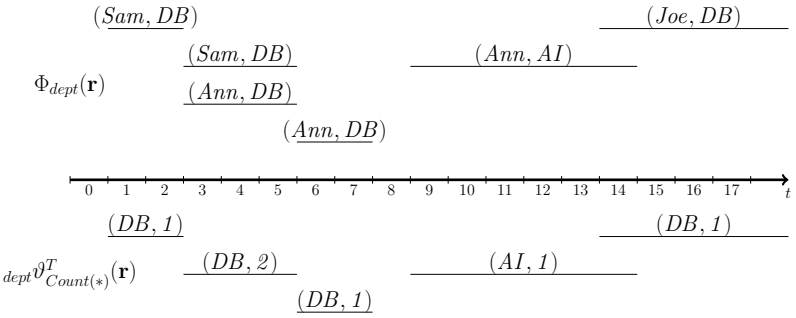


Figure 5.4: Temporal Aggregation Reduced to Non-Temporal Aggregation.

5.3 Difference

The temporal difference is a binary operator which subtracts from the first relation the tuples for which in the second relation a value-equivalent tuple at the same time-point exists.

Definition 5. The *temporal difference* between two temporal relations $\mathbf{r}$ and $\mathbf{s}$

is defined as

$$\mathbf{r} -^T \mathbf{s} = \{z \mid \exists r \in \mathbf{r}(z[A] = r[A] \wedge z.T \subseteq r.T \wedge \\ \forall s \in \mathbf{s}(s[A] = z[A] \Rightarrow s.T \cap z.T = \emptyset) \wedge \\ \forall T \supset z.T \exists s \in \mathbf{s}(s[A] = z[A] \wedge s.T \cap T \neq \emptyset \vee T \not\subseteq r.T))\}$$

The temporal difference contains for each tuple, $r \in \mathbf{r}$, a result tuple over all maximal sub-intervals of $r.T$, which are not covered by a value-equivalent tuple in $\mathbf{s}$.

Example 8. Figure 5.5 illustrates the temporal difference for our running example. For instance, the result tuple $(\text{Sam}, \text{DB}, [1, 3])$ is produced from r_1 over the (maximal) time period that is not covered by any tuple in $\mathbf{s}$ with the same name and department values.

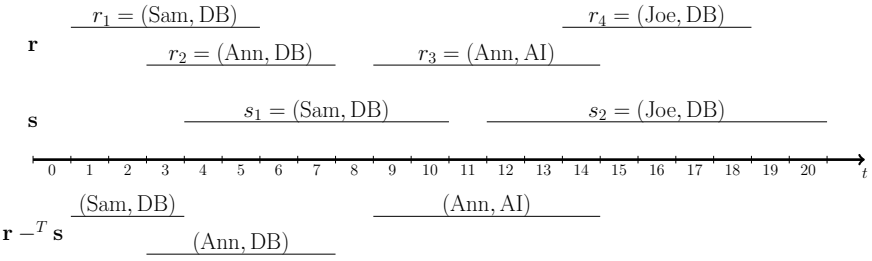


Figure 5.5: Temporal Difference.

The following theorem shows how, by using binary temporal unification, the temporal difference operator can be reduced to the non-temporal difference.

Theorem 5. *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be two temporal relations with non-temporal attributes* A *and timestamp attribute* T*. The temporal difference between* $\mathbf{r}$ *and* $\mathbf{s}$ *can be reduced to the non-temporal difference as follows:*

$$\mathbf{r} -^T \mathbf{s} \equiv (\mathbf{r}\Phi_{r[A]=s[A]}\mathbf{s}) - (\mathbf{s}\Phi_{r[A]=s[A]}\mathbf{r})$$

Proof. We show the equivalence of the sets produced by the right-hand side of the theorem and the definition of the temporal difference. Let $\mathbf{z}_r$ be the result of $\mathbf{r}\Phi_{r[A]=s[A]}\mathbf{s}$. Then $\mathbf{z}_r$ can be partitioned into $\mathbf{z}'_r$ and $\mathbf{z}''_r$, where $\mathbf{z}'_r$ is produced by expression 1 and $\mathbf{z}''_r$ by expression 2 of Def. 2. In a similar way, we have $\mathbf{z}_s = \mathbf{s}\Phi_{r[A]=s[A]}\mathbf{r}$, which can be partitioned into $\mathbf{z}'_s$ and $\mathbf{z}''_s$. Now we have to show that $\mathbf{r} -^T \mathbf{s} = (\mathbf{z}'_r \cup \mathbf{z}''_r) - (\mathbf{z}'_s \cup \mathbf{z}''_s)$. First, we show that $\mathbf{z}'_r = \mathbf{z}'_s$. By substituting θ in Def. 2 with $r[A] = s[A]$, the expressions 1 that produce these two sets become identical and the equivalence follows immediately. Thus, $\mathbf{z}_r$ will not be in the result of the non-temporal difference. Since $\mathbf{z}'_s$ and $\mathbf{z}''_s$ are disjoint and $\mathbf{z}'_r = \mathbf{z}'_s$ we have that $\mathbf{z}'_r$ and $\mathbf{z}''_s$ are disjoint, too, and we get $\mathbf{r} -^T \mathbf{s} = \mathbf{z}''_r - \mathbf{z}''_s$. Second, we show that the sets $\mathbf{z}''_r$ and $\mathbf{z}''_s$ are disjoint by showing that the corresponding expressions 2 in Def. 2 cannot be satisfied in

conjunction, i.e.,

$$\begin{aligned}
&\exists r \in \mathbf{r}(z[A] = r[A] \wedge z.T \subseteq r.T \wedge \\
&\qquad \nexists s \in S(r[A] = s[A] \wedge s.T \cap z.T \neq \emptyset) \\
&\quad \wedge \\
&\exists s \in \mathbf{s}(z[A] = s[A] \wedge z.T \subseteq s.T \wedge \\
&\qquad \nexists r \in \mathbf{r}(r[A] = s[A] \wedge r.T \cap z.T \neq \emptyset)
\end{aligned}$$

is always false, which can easily be seen. Finally, we get $\mathbf{r} -^T \mathbf{s} = \mathbf{z}''_r$, and the expression that produces $\mathbf{z}''_r$ is identical to the definition of the temporal difference, i.e., $\exists r \in \mathbf{r}(z[A] = r[A] \wedge z.T \subseteq r.T \wedge \forall s \in \mathbf{s}(r[A] \neq s[A] \vee s.T \cap z.T = \emptyset)$ and the intervals are maximal. □

Example 9. Figure 5.6 illustrates the computation of the temporal difference using Theorem 5. Notice that for this operation the binary unification has to be applied in both directions. Then the non-temporal difference between the two unification result determines the intended result of temporal aggregation.

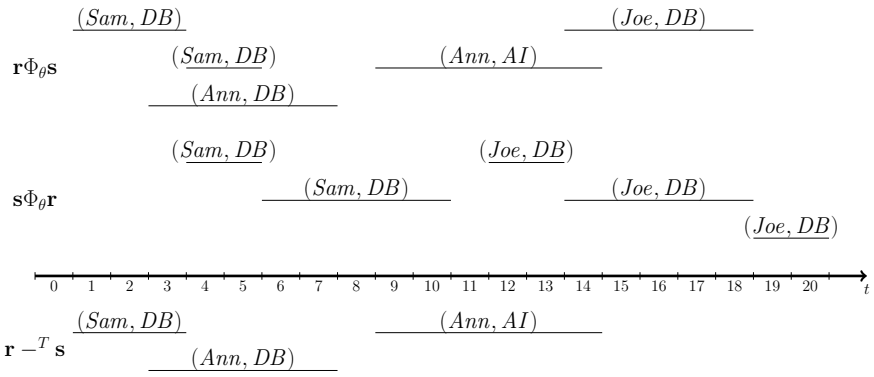


Figure 5.6: Temporal Difference Reduced to Non-Temporal Difference.

5.4 Intersection

The temporal set intersection is a binary operator which retains all tuples of one relation for which in the other relation a value-equivalent tuple at the same time-point exists.

Definition 6. The *temporal intersection* between two temporal relation **r** and **s** is defined as

$$\begin{aligned}
\mathbf{r} \cap^T \mathbf{s} = \{z \mid \exists r \in \mathbf{r}(z[A] = r[A] \wedge \\
\exists s \in \mathbf{s}(r[A] = s[A] \wedge z.T = r.T \cap s.T \wedge z.T \neq \emptyset))\}
\end{aligned}$$

The temporal intersection contains for each tuple $r \in \mathbf{r}$ the sub-interval of $r.T$ which is completely covered by a value equivalent tuple in **s**.

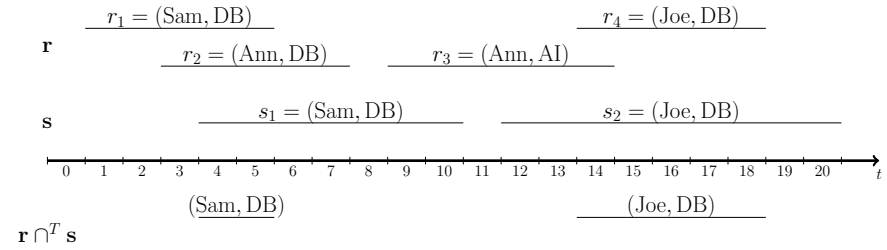


Figure 5.7: Temporal Intersection.

Example 10. Figure 5.7 illustrates the temporal intersection for our running example. For instance, the result tuple $(\text{Sam}, \text{DB}, [4, 5])$ is produced from the intersection of the tuples r_1 and s_1.

The following theorem shows how, by using binary temporal unification, the temporal intersection operator can be reduced to the non-temporal intersection.

Theorem 6. *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be two temporal relations with non-temporal attributes* A *and timestamp attribute* T*. The temporal intersection between* $\mathbf{r}$ *and* $\mathbf{s}$ *can be reduced to non-temporal intersection as follows:*

$$\mathbf{r} \cap^T \mathbf{s} \equiv (\mathbf{r}\Phi_{r[A]=s[A]}\mathbf{s}) \cap (\mathbf{s}\Phi_{r[A]=s[A]}\mathbf{r})$$

Proof. The proof for temporal temporal intersection is similar to the proof of Theorem 5. In this case we need to show that $\mathbf{r} \cap^T \mathbf{s} = (\mathbf{z}'_r \cup \mathbf{z}''_r) \cap (\mathbf{z}'_s \cup \mathbf{z}''_s)$. From the reasoning of the previous proof we know that $\mathbf{z}'_r = \mathbf{z}''_r$, when θ is the equality predicate $r[A] = s[A]$, hence the non-temporal intersection on the right-hand side will retain the set $\mathbf{z}'_r$. Further, we know that $\mathbf{z}''_r$ and $\mathbf{z}''_r$ are disjoint, so we get $\mathbf{r} \cap^T \mathbf{s} = \mathbf{z}'_r$. Finally, we have that the expression that produces $\mathbf{z}'_r$ is identical to the definition of the temporal intersection, i.e., $\exists r \in \mathbf{r} \exists s \in \mathbf{s}(r[A] = s[A] \wedge z[A] = r[A] \wedge z.T = r.T \cap s.T \wedge z.T \neq \emptyset))$ □

Example 11. Figure 5.8 illustrates the computation of the temporal intersection using Theorem 6. As for the other set operations, the unification is required in both directions. Then the non-temporal intersection between the two unified relations determines the intended result of the temporal intersection.

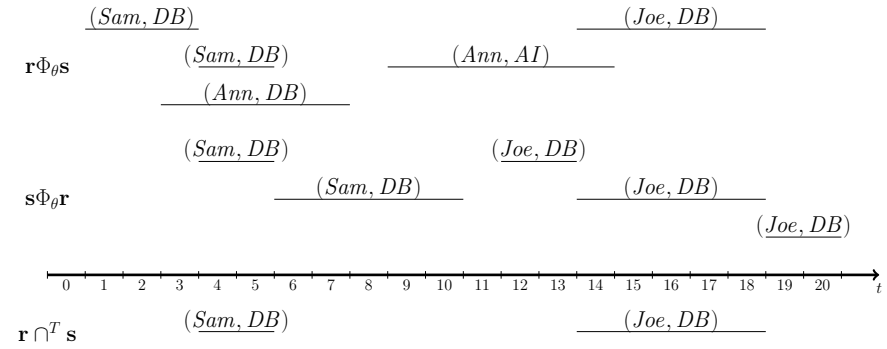


Figure 5.8: Temporal Intersection Reduced to Non-Temporal Intersection.

5.5 Union

The temporal set union is a binary operator which retains the data of both input relations. This operator, like all other set operations, is designed to produce sets, i.e., duplicates are not retained in the result.

Definition 7. The *temporal union* between two temporal relation, $\mathbf{r}$ and $\mathbf{s}$, is defined as

$$
\begin{aligned}
\mathbf{r} \cup^T \mathbf{s} = \{z \mid\ & \exists r \in \mathbf{r}(z[A] = r[A] \wedge z.T \subseteq r.T \wedge \\
& \quad \forall s \in \mathbf{s}(s[A] = z[A] \Rightarrow s.T \cap z.T = \emptyset) \wedge \\
& \quad \forall T \supset z.T \exists s \in \mathbf{s}(s[A] = z[A] \wedge s.T \cap T \neq \emptyset \vee T \not\subseteq r.T)) \\
& \quad \vee \\
& \exists r \in \mathbf{r}(z[A] = r[A] \wedge \\
& \quad \exists s \in \mathbf{s}(r[A] = s[A] \wedge z.T = r.T \cap s.T)) \\
& \quad \vee \\
& \exists s \in \mathbf{s}(z[A] = s[A] \wedge z.T \subseteq s.T \wedge \\
& \quad \forall r \in \mathbf{s}(r[A] = z[A] \Rightarrow r.T \cap z.T = \emptyset) \wedge \\
& \quad \forall T \supset z.T \exists r \in \mathbf{r}(r[A] = z[A] \wedge r.T \cap T \neq \emptyset \vee T \not\subseteq s.T))\}
\end{aligned}
$$

The temporal union contains for each tuple $r \in \mathbf{r}$ and $s \in \mathbf{s}$ all those maximal sub-intervals, which are not covered by a value-equivalent tuple in the other relation. Furthermore, it contains for each tuple $r \in \mathbf{r}$ the sub-interval which is completely covered by a value-equivalent tuple $s \in \mathbf{s}$. This expression makes sure that the result does not contain duplicates.

Example 12. Figure 5.9 illustrates the temporal union for our running example. For instance, the result tuple $(\text{Sam}, \text{DB}, [1, 3])$ is produced from the tuple r_1 and the tuple $(\text{Sam}, \text{DB}, [4, 5])$ from the duplicate elimination of tuples r_1 and s_1 over that sub-interval.

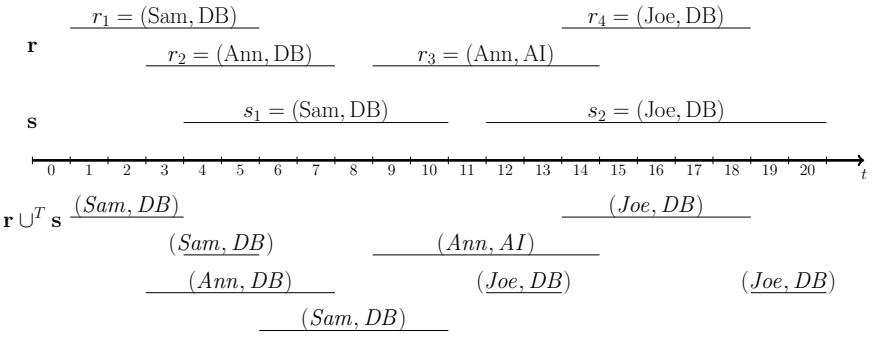


Figure 5.9: Temporal Union.

The following theorem shows how, by using binary temporal unification, the temporal union operator can be reduced to the non-temporal union.

Theorem 7. *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be two temporal relations with non-temporal attributes* A *and timestamp attribute* T*. The temporal union between* $\mathbf{r}$ *and* $\mathbf{s}$ *can be reduced to non-temporal union as follows:*

$$\mathbf{r} \cup^T \mathbf{s} \equiv (\mathbf{r}\Phi_{r[A]=s[A]}\mathbf{s}) \cup (\mathbf{s}\Phi_{r[A]=s[A]}\mathbf{r})$$

Proof. To proof this theorem, we apply the same procedure as for the previous proofs. We show that the reduction rule $\mathbf{r} \cup^T \mathbf{s} \equiv (\mathbf{r}\Phi_{r[A]=s[A]}\mathbf{s}) \cup (\mathbf{s}\Phi_{r[A]=s[A]}\mathbf{r})$ is correct by showing that $\mathbf{r} \cup^T \mathbf{s} = (\mathbf{z}'_r \cup \mathbf{z}''_r) \cup (\mathbf{z}'_s \cup \mathbf{z}''_s)$. We know that $\mathbf{z}''_r$ is disjoint from all other involved sets, and therefore it will be retained in the result of the non-temporal union. The same holds for the set $\mathbf{z}''_s$, which is also retained in the result. Since $\mathbf{z}'_r = \mathbf{z}'_s$, the non-temporal union will retain only one of them (e.g., $\mathbf{z}'_r$) in the result, and we get $\mathbf{r} \cup^T \mathbf{s} = \mathbf{z}''_r \cup \mathbf{z}'_r \cup \mathbf{z}''_s$. Finally, by substituting $\mathbf{z}''_r$, $\mathbf{z}'_r$, and $\mathbf{z}'_r$ with the expression producing them in a disjunction, the right-hand side becomes identical to the definition of temporal union. □

Example 13. Figure 5.10 illustrates the computation of the temporal intersection using Theorem 7. After applying binary unification, the set union of the two unification results produces the intended result. Note that the tuple $\text{Sam}, \text{DB}, [4, 6)$, which appears in both unification results, appears only once in the final result (due to the set semantics of the non-temporal union).

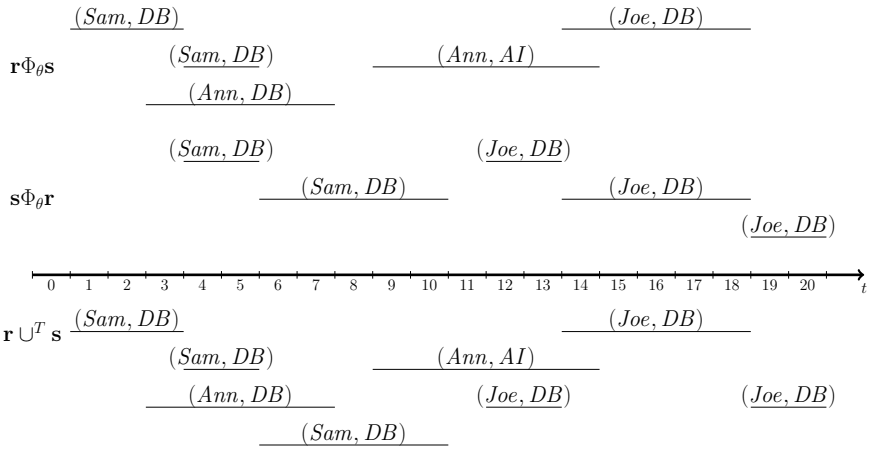


Figure 5.10: Temporal Union Reduced to Non-Temporal Union.

5.6 Inner Join

The temporal inner join is a binary operator that uses a boolean predicate, θ, as a join-condition over the non-temporal attributes of the two input relations. The result of the temporal join contains all pairs of tuples from the first and the second argument relation that satisfy θ, ranging over the common temporal sub-interval.

Definition 8. The *temporal inner join* between two temporal relation $\mathbf{r}$ and $\mathbf{s}$

using join-condition θ is defined as

$$\mathbf{r} \bowtie^T_\theta \mathbf{s} = \{z \mid \exists r \in \mathbf{r} s \in \mathbf{s}(z[A] = r[A] \circ s[A] \wedge \theta(r,s) \wedge z.T = r.T \cap s.T \wedge z.T \neq \emptyset)\}$$

The temporal join contains the concatenation of all r- and s-tuples that satisfy the join predicate θ and which are temporally overlapping; the non-empty intersection of the two timestamps determines the timestamp of the result tuple.

Example 14. Figure 5.11 illustrates the temporal inner join for our running example, when θ is an equality predicate between the corresponding *Dept* attributes.

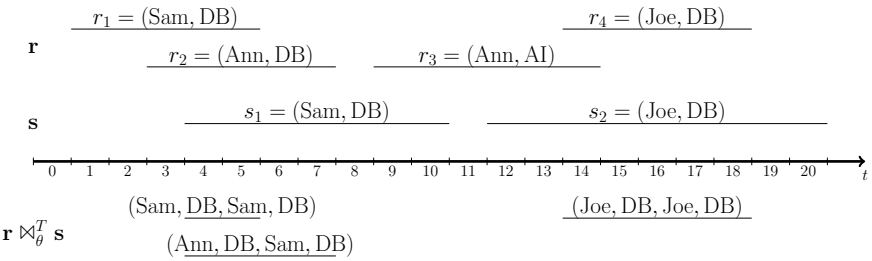


Figure 5.11: Temporal Inner Join.

The following theorem shows how, by using binary temporal unification, the temporal union operator can be reduced to the non-temporal union.

Theorem 8. *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be two temporal relations with non-temporal attributes* A *and timestamp attribute* T*. The temporal join between* $\mathbf{r}$ *and* $\mathbf{s}$ *can be reduced to the non-temporal join as follows:*

$$\mathbf{r} \bowtie^T_\theta \mathbf{s} \equiv (\mathbf{r}\Phi_\theta\mathbf{s}) \bowtie_{\theta \wedge \mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$$

Proof. To proof the reduction of the temporal join we apply the same strategy as for the set operations. However, for the general θ-join we have that $z_r = \mathbf{r}\Phi_\theta\mathbf{s}$ and $z_s = \mathbf{s}\Phi_\theta\mathbf{r}$. As before, z_r is partitioned into z'_r and z''_r on the expressions 1 and 2, respectively. Similarly, z'_s and z''_s are the partitions of z_s. We then show that $\mathbf{r} \bowtie^T_\theta \mathbf{s} = (\mathbf{z}'_r \cup \mathbf{z}''_r) \bowtie_{\theta \wedge r.T=s.T} (\mathbf{z}'_s \cup \mathbf{z}''_s)$, where $r.T$ is the timestamp attribute of the set $(\mathbf{z}'_r \cup \mathbf{z}''_r)$ and $s.T$ is the timestamp attribute of the set $(\mathbf{z}'_s \cup \mathbf{z}''_s)$. First, we show that the sets $\mathbf{z}''_r$ and $\mathbf{z}''_s$ do not contain tuples that satisfy $\theta(z''_r, z''_s)$ and have equal timestamp attributes. By considering the expression 2 in Def. 2, which produces the two sets, we show that they are not satisfiable in conjunction:

$$\begin{aligned}
&\theta(z''_r, z''_s) \wedge z''_r.T = z''_s.T \wedge \\
&\exists r \in \mathbf{r}(z''_r[A] = r[A] \wedge z''_r.T \subseteq r.T \wedge \\
&\qquad \nexists s \in S(\theta(r,s) \wedge s.T \cap z''_r.T \neq \emptyset) \\
&\qquad \wedge \\
&\exists s \in \mathbf{s}(z''_s[A] = s[A] \wedge z''_s.T \subseteq s.T \wedge \\
&\qquad \nexists r \in \mathbf{r}(\theta(r,s) \wedge r.T \cap z''_s.T \neq \emptyset)
\end{aligned}$$

Second, we show that the sets $\mathbf{z}'_r$ and $\mathbf{z}''_s$ do not contain tuples that satisfy $\theta(z'_r, z''_s)$ and have equal timestamp attributes, as follows:

$$\begin{aligned}
&\theta(z'_r, z''_s) \wedge z'_r.T = z''_s.T \wedge \\
&\quad \exists r \in \mathbf{r} \exists s \in \mathbf{s}(\theta(r,s) \wedge z'_r[A] = r[A] \wedge \\
&\qquad z'_r.T = r.T \cap s.T \wedge z'_r.T \neq \emptyset) \\
&\qquad \wedge \\
&\quad \exists s \in \mathbf{s}(\mathbf{z}''_s[A] = s[A] \wedge \mathbf{z}''_s.T \subseteq s.T \wedge \\
&\qquad \nexists r \in \mathbf{r}(\theta(r,s) \wedge r.T \cap \mathbf{z}''_s.T \neq \emptyset)
\end{aligned}$$

Similarly, we can show the same for $\mathbf{z}''_r$ and $\mathbf{z}'_s$. The non-temporal θ-join will therefore only match tuples from the sets $\mathbf{z}'_r$ and $\mathbf{z}'_s$, and we get $\mathbf{r} \bowtie^T_\theta \mathbf{s} = \mathbf{z}'_r \bowtie_{\theta \wedge r.T = s.T} \mathbf{z}'_s$. Next, we can insert the expressions from which $\mathbf{z}'_r$ and $\mathbf{z}'_s$ are produced into the non-temporal join expression, and we get

$$\begin{aligned}
\mathbf{r} \bowtie^T_\theta \mathbf{s} \equiv \{z \mid z = z'_r \circ z'_s \wedge z'_r.T = z'_s.T \wedge \\
\exists r \in \mathbf{r} \exists s \in \mathbf{s}(\theta(r,s) \wedge z'_r[A] = r[A] \wedge z'_s[A] = s[A] \wedge \\
z'_r.T = r.T \cap s.T \wedge z'_s.T = r.T \cap s.T)\}
\end{aligned}$$

Note that both expressions can be merged into a single exists clause, since we do not add any restrictions to the variables of each expression. It is now possible to see that the above expression coincides with the definition of the temporal θ-join. To have exact correspondence although, the reduction using binary temporal unification needs an additional projection in order to eliminate the duplicate timestamp attributes caused by the concatenation of attributes, if those are retained by the non-temporal join. □

Example 15. Figure 5.12 illustrates the computation of the temporal inner join using Theorem 8. Notice that the non-temporal join treats the timestamp attributes as a non-temporal attribute and compares them using equality.

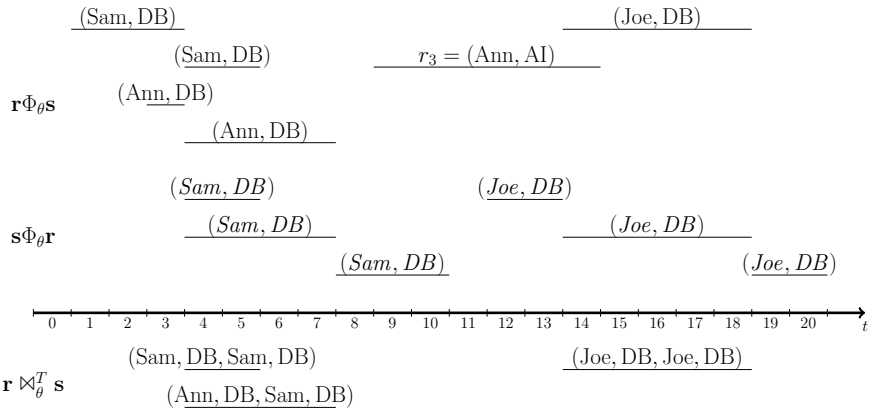


Figure 5.12: Temporal Join Reduced to Non-Temporal Join.

The temporal Cartesian product can be produced using the same procedure by using *true* as the θ condition.

5.7 Outer Join

The temporal left outer join is a binary operator using a boolean predicate θ, where θ is a join-condition over the non-temporal attributes of the input relations. The result of the temporal join are all combinations of non-temporal attributes of the first and second argument relation that satisfy θ, ranging over the common time sub-interval. All maximal sub-intervals of tuples of the first input relation that are not covered by any tuple in the second relation but matching θ are retained in the result, with $NULL$ values in place of the second relation attribute values.

There exist two other forms of temporal outer joins, one is the right outer join being the symmetric counterpart to the left join. The third form of outer join is the full outer join, which is the combination of both.

Definition 9. The *temporal left join* between two temporal relation $\mathbf{r}$ and $\mathbf{s}$ using join-condition θ is defined as

$$\begin{aligned}\mathbf{r} \,⟕^T_\theta \mathbf{s} = \{z \mid \exists r \in \mathbf{r}(\exists s \in \mathbf{s}(z[A] = r[A] \circ s[A] \wedge \theta(r,s) \wedge z.T = r.T \cap s.T \vee \\ z[A] = r[A] \circ (\bot, \ldots, \bot) \wedge z.T \subseteq r.T \wedge \\ \forall s \in \mathbf{s}(\neg\theta(r,s) \vee s.T \cap z.T = \emptyset) \wedge \\ \forall T \supset z.T \exists s \in \mathbf{s}(\theta(r,s) \wedge s.T \cap T \neq \emptyset \vee T \not\subseteq r.T))\}\end{aligned}$$

The temporal left join contains each concatenation of r- and s-tuples that satisfy the join predicate θ and which are temporally overlapping; the overlapping part is the timestamp of the result tuple. For each maximal sub-interval T of a tuple $r \in \mathbf{r}$, which is not covered by a tuple $s \in \mathbf{s}$ that satisfies θ, the result contains a combination of the form $r[A] \circ (\bot, \ldots, \bot)$ and T, where $(\bot, \ldots, \bot)$ are $NULL$ values in place of the missing $s[A]$.

Example 16. Figure 5.13 illustrates the temporal left outer join for our running example when θ is an equality predicate over the *Dept* attributes.

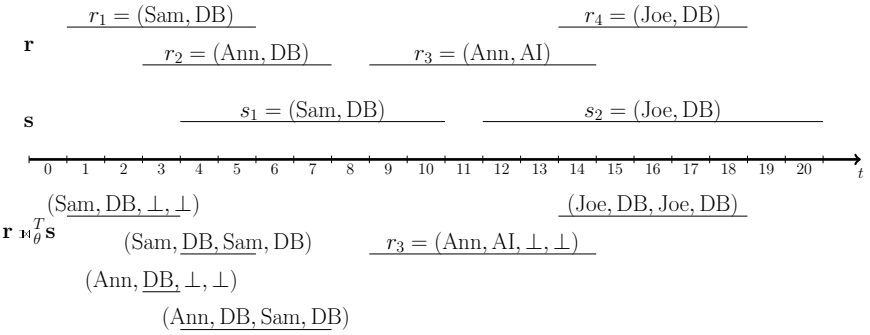


Figure 5.13: Temporal Left Outer Join.

The following theorem shows how, by using binary temporal unification, the temporal union operator can be reduced to the non-temporal union.

Theorem 9. *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be two temporal relations with non-temporal attributes* A *and timestamp attribute* T*. The temporal left outer join between* $\mathbf{r}$ *and* $\mathbf{s}$ *can*

be reduced to non-temporal left outer join as follows:

$$\mathbf{r} \overset{T}{\rtimes}_{\theta} \mathbf{s} \equiv (\mathbf{r}\Phi_{\theta}\mathbf{s}) \rtimes_{\theta \wedge \mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_{\theta}\mathbf{r})$$

Proof. The proof for the correctness of reducing the temporal left join to the non-temporal left join using binary unification is similar to the proof of Theorem 8. The only difference is that the set z_r'' is retained by the non-temporal left join, and since those tuples in $\mathbf{r}$ do not match any tuple in $\mathbf{s}$, their join is concatenated with $NULL$ values. Then, we get the right hand side of the theorem corresponding to the definition of the temporal left outer join. □

Example 17. Figure 5.14 illustrates the computation of the temporal left outer join using Theorem 9.

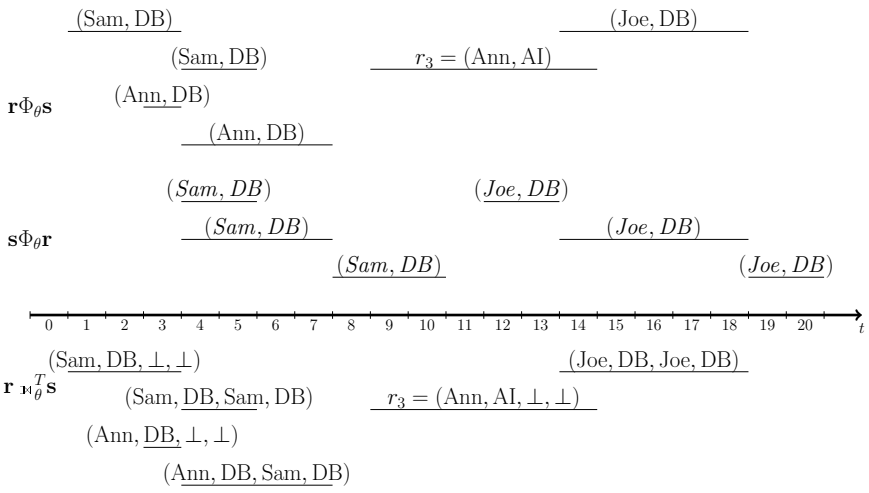


Figure 5.14: Temporal Left Join Reduced to Non-Temporal Left Join.

The other temporal outer joins as right and full outer join can similarly be reduced to their non-temporal corresponding outer join.s

Chapter 6

Implementation

In this section we describe an implementation of the temporal algebra in the PostgreSQL database management system. For the temporal unification two new operators have been implemented for unary and binary unification. The individual temporal operators are then realized using the reduction rules discussed in the previous chapter. Such a solution significantly increases the efficiency of the temporal operators compared to middle-ware solutions, since middle-ware solutions need to fetch the data from the database server over sockets. The implementation was done using version 8.4.2 of PostgreSQL.

6.1 The PostgreSQL Query Flow Model

The PostgreSQL database system adopts a client-server architecture. The client connects to the database server over a socket and communicates with the server following a specific protocol. The result computed by the database server is then send back to the client via a network socket. The actual work of the database server is between these two points: a query is requested by the client and the query result is returned to the client.

To answer a SQL query requested by the client, the PostgreSQL server follows a well-defined flow of control from the point the query is issued to the point when specific algorithms are executed to manipulate the stored data according to the user query in order to generate the final query result. Figure 6.1 shows the most important steps of this workflow. Each stage has a well defined input and produces a well defined output. The output of the last stage is the answer to the query request. In the following, these stages are described in more detail.

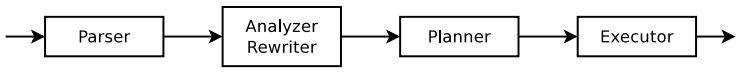


Figure 6.1: PostgreSQL Query Flow.

6.1.1 Parser

The first stage in query processing is to parse the input query. The query is sent to the database server as a string, i.e., a sequence of characters, which

need to be checked for syntactic correctness. This is done in the parsing phase. The parser in PostgreSQL is implemented using LEX and YACC, where the former is a lexical analyzer and the latter is a parser generator for context free grammars.

The parser transforms the string sent by the client into a so-called *Parse Tree*, which is stored in a specific C-struct, named `ParseTree`, and represents the output of the parsing stage. The output of this stage is a tree, since SQL allows nested queries, i.e., an item in the `FROM` clause can be a simple relation or a sub-query. Such recursive structures can easily be represented in tree structures. The ParseTree is then passed to the next stage.

6.1.2 Analyzer and Rewriter

The analyzer receives the parse tree from the Parser as input and performs various checks on it. For instance, it must ensure that all specified items in the query are actually present in the database system, i.e., all specified relations exist and the user is allowed to access them. Furthermore, a check is performed to verify that all columns referenced in the query exist and whether the operators specified in the query can be applied to these columns.

The analyzer is implemented recursively. It starts at the bottom of the parse tree, and in a bottom up approach it incrementally constructs a query tree. The *query tree* is stored in a C-struct, named `QueryTree`, and has the same structure as the parse tree, but it contains additional information which is needed for further analysis in the following stages. The main difference between the parse tree and the query tree is that in the query tree all columns of the involved relations are explicitly specified, e.g., a $\star$ in the `SELECT` clause is expanded to the actual column names, including additional information such as data types.

The rewriter in this combined phase performs modifications on the query tree by adding further information, most importantly by replacing SQL views with their definition in order for the next steps to be view independent. The structure of the tree is not changed.

6.1.3 Planner

The task of the planner is to find an optimal plan for the query execution. It takes as input the query tree from the previous stage and tries to find the most efficient way to execute the given query. Similar to the analyzer, the planner operates recursively, starting at the bottom of the query tree and generating paths which are suitable to execute. In a path the algebraic operators are replaced by specific algorithms together with the required parameters and the estimated execution cost. For instance, a join can be replaced by a nested loop join or a merge join, where the latter might require an additional sorting step. When more paths are available, the planner chooses the one with the lowest estimated cost.

The PostgreSQL query planner especially focuses on ordered outputs. Some algorithms, such as merge join or aggregation/grouping, produce sorted results, which might be helpful for the choice of the algorithm that manipulates the data next.

The planner returns a *plan tree*, which is stored in a C-struct, named `PlanTree`. In the plan tree all operators are replaced by specific algorithms

together with relevant parameters and an the expected cost. A part of the information stored in the plan tree can be retrieved by issuing the SQL `Explain` command.

6.1.4 Executor

The executor stage takes as input the plan tree struct from the previous step and makes it ready for execution. Each node in the plan tree specifies an algorithm, where each algorithm is realized by three functions, which we describe next; `Algo` is a placeholder for the actual name of an algorithm.

`ExecInitAlgo.` This function is called before the actual algorithm is executed. It takes as input a node of the plan tree. The function performs all initializations for an algorithm as well as for all relevant sub-nodes. The return value of this function is a C-struct, named `AlgoState`, where the algorithm stores state information during its execution; this struct is passed to all remaining functions.

`ExecAlgo.` This function implements the execution of an algorithm. Its takes as input the current state information `AlgoState` returned by the initialization function. The output of this function is either a single result tuple or `NULL`, which indicates the termination of the algorithm. The struct `AlgoState` can be used to retrieve tuples from the sub-nodes of the current algorithm and to store context information, which will be available the next time this function is called.

`ExecEndAlgo.` This function performs clean-up tasks such as releasing the memory that was allocated during the initialization and execution stage. The `ExecEndAlgo` functions are recursively called for all sub-nodes of the current node.

6.2 Unary Unification

6.2.1 Overview

The implementation of the unary unification operator in PostgreSQL require to go through all steps of the previously described query flow, from the definition of a grammar in the parsing step to the final execution of the specific algorithm. Thereby, we tried to reuse existing code as much as possible.

The most critical point for the implementation of unary unification is to avoid the nested loop in the algorithm in Figure 4.3, which finds for each tuple r in the argument relation all other tuples which have identical values for the attributes B and whose either start or end timestamp falls into the interval of r. This can be achieved by re-using an (non-temporal) internal join provided by the DBMS. Then the result of this join is scanned and all time points are processed. The drawback of such a solution would be that all time points need to be stored in memory, since they have to be processed in chronological order. This although is not feasible, as to much main memory could be required.

Since we are only interested in the time points and not in the intervals of each matching tuple s, we could alternatively first perform the union of all start

and end points, including the attributes B, and then join the relation $\mathbf{r}$ with the result. Afterwards, we have to sort according to all attributes from the left part of the join and the time point of the right part, which allows us to process the time points in chronological order. Thus, the query to realize the unification operation $\Phi_B(\mathbf{r})$ is as follows: $\mathbf{r} \bowtie_\theta (\pi_{B,T_S}(\mathbf{r}) \cup \pi_{B,T_E}(\mathbf{r}))$, where θ is an equality predicate over the attributes B of the left and right join expression and the containment of the right time point in the left time interval. Following this join, we apply a projection to remove one instance of the attributes B, due to the join they are duplicated in the schema.

Following the above strategy, the implementation of the unary unification algorithm has been divided into three steps:

1. perform a non-temporal Left Join to produce $\mathbf{z}'$;
2. sort $\mathbf{z}'$;
3. produce the result of unary unification while scanning $\mathbf{z}'$.

Figure 6.2 illustrates step 3 with three tuples, z'_1, z'_2, and z'_3. Each tuple is in the result of the join, where everything except the last attribute comes from the left relation of the join, denoted as r_x. The last attribute of the join result is a time point, denoted as t_x. The relation is sorted according to all attributes. Now we start to produce result tuples of the unary unification. When reading the first tuple, z'_1, a tuple from the start point of r_1 till the time point t_1 is produced (since we know that the time points are sorted chronologically). Next, we store tuple z'_1 and read z'_2. Since both tuples have the same left part (r_1), we produce a new tuple from time point t_1 to time point t_2. The tuple z'_1 is replaced by z'_2. Then, z'_3 is scanned. Since z'_3 has a different left part (r_2), we can finish the previous tuple by producing a new tuple from t_2 to the end time of r_1. Then we proceed with z'_3 as we did for z'_1.

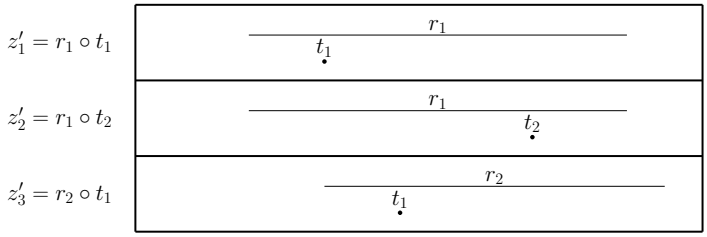


Figure 6.2: Unary Unification from Left Join.

Example 18. Consider the unary unification $\Phi_{Dept}(\mathbf{r})$ using our running example. First, we build the join expression $\mathbf{r}/\mathbf{r}_l \bowtie_\theta (\pi_{Dept,T_S/P}(\mathbf{r}) \cup \pi_{Dept,T_E/P}(\mathbf{r}))/\mathbf{r}_r$, where / is the rename operator and θ is and equality predicate over the left and right *Dept* attributes and the containment of the right time point in the left time interval, i.e., $\theta \equiv \mathbf{r}_l.Dept = \mathbf{r}_r.Dept \wedge \mathbf{r}_r.P > \mathbf{r}_l.T_S \wedge \mathbf{r}_r.P < \mathbf{r}_l.T_E$. Then we project the result according to $\mathbf{r}_l.Dept, \mathbf{r}_l.T, \mathbf{r}_r.P$, to remove the duplicated *Dept* attribute resulting from $\mathbf{r}_l$. The result of this expression is shown in Table 6.1.

The next step is to sort the result according to all attributes. This guarantees that joins for each original left tuple (r_l) are consecutive and the time points

	r_l			r_r
	Emp	*Dept*	T	P
j_1	Ann	AI	$[9, 15)$	$\perp$
j_2	Ann	DB	$[3, 8)$	6
j_3	Joe	DB	$[14, 19)$	$\perp$
j_4	Sam	DB	$[1, 6)$	3

Table 6.1: Join Result to compute Unary Unification.

P of the original right tuple (r_r) are sorted in chronological order. We can see that the result tuple j_1 has a `NULL` value for attribute P. That is, there is no joining tuple and therefore the tuple corresponding to r_l can be added to the result, which is shown as tuple u_1 in Table 6.2. The tuple j_2 has a value for P, so we produce the tuple u_2. Since no more joins for this r_l exist (j_3 has different attribute values for r_l), we add u_3 as a second result tuple.

	Emp	*Dept*	T
u_1	Ann	AI	$[9, 15)$
u_2	Ann	DB	$[3, 6)$
u_3	Ann	DB	$[5, 8)$
u_4	Joe	DB	$[14, 19)$
u_5	Sam	DB	$[1, 3)$
u_6	Sam	DB	$[3, 6)$

Table 6.2: Result of $\Phi_{\mathbf{r}}(Dept)$.

In the following we describe in detail the individual steps of the implementation of the unary unification algorithm.

6.2.2 Parser

To integrate the unary unification operator into the SQL language we introduce a new keyword `UNIFY`, which requires a relation and a list of attributes as input. The required modifications in PostgreSQL's grammar file `gram.y` are as follows:

```
table_ref:
        UNIFY table_ref USING '(' name_list ')'
        { ... };
        | UNIFY table_ref USING '(' ')'
        { ... };
```

Two new roles for the operator are introduced, where at the end the operator reduces to a `table_ref`, which is an item of SQL's `FROM` clause. Thus, the operator can be used as a conventional relation in the language. The operator starts by specifying the keyword `UNIFY`, followed by a relation or sub-query, then the keyword `USING` and a list of comma-separated attributes in parentheses. Since a list of attributes defined in PostgreSQL cannot be empty, a second rule covers the case when the operator gets the empty set of attributes to unify. As an example of the new grammar, the SQL statement for the expression $\Phi_{Dept}(\mathbf{r})$ is as follows:

```
Select emp, dept, ts, te
From Unify r
     Using (dept);
```

When the parser reduces to the unary unification rules, a new C-struct is created, called `UUnifyExr`, which represents a new node in a parse tree. The C-struct `UUnifyExr` is implemented as follows:

```
struct UUnifyExpr
{
  NodeTag   type; /* type of this struct */
  Node     *arg; /* argument subtree */
  List     *using; /* list of attributes to unify, if any */
}
```

The first variable is a type tag and is required for PostgreSQL's pseudo object orientation in order to recognize the type of a node (i.e., the type of the struct). The type variable is initialized to `T_UUnifyExpr`. The second variable can store a general node, which in this case represents the argument relation, e.g., a relation name or a sub-statement. As a third variable the list of attributes to unify is stored; this variable can be `NIL`, which indicates that no attribute is passed to the operator.

6.2.3 Analyzer and Rewriter

The analysis phase gets the parse tree as input and transforms it into a query tree. The newly created `UUnifyExr` of the parse tree has to be analyzed and transformed to a node of the query tree structure.

As first step, the underlying join sub-statement is generated from the given information, which is then inserted as the new argument of the unification statement. By creating this sub-statement, the analysis can be done by the join subnode, since it performs checks if the argument relation exists and if all attributes are present and accessible by the user. So no more work for the analysis stage has to be performed. This is also true for the rewriting stage. The argument passed to the unification operator is directly passed to the join sub-node, which takes care that views are correctly rewritten.

6.2.4 Planner

In this stage, the unary unification statement has to be replaced by a planning node, which represents the unary unification algorithm. As before, our statement now has only one argument relation, which is the join sub-statement generated before. The planner operates recursively. When planning the unification node, the information from the sub-statement is already available, i.e., information about the assumed number of rows returned by the sub-statement, the estimated cost, and whether the data is sorted according to some attributes. For the unary unification operator, we have to approximate the same information an deliver it to the planner.

To approximate the number of rows produce by the unary unification algorithm, we use a simple formula, which calculates the maximum number of rows the algorithm can produce. By scanning the input from the sub node, for each

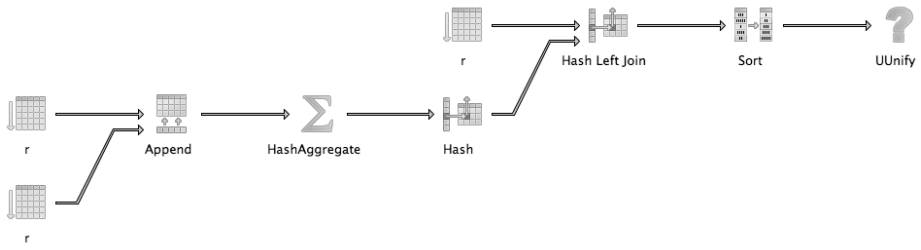


Figure 6.3: Example Explain Unary Unify.

tuple of the input at most two tuples can be produced, so we can use the information from the sub-node about the approximated number of rows produced by it and multiply it by a factor of 2. This information is stored in the planning node.

For the cost approximation we use also a simple formula, which calculates the cost of comparing each tuple with the next. Therefore, the cost of the algorithm is approximated as $cost = cpu\_operator\_cost * numRows * numCols$, where *cpu_operator_cost* is the cost unit of one cpu operation, $numRows$ is the approximated number of returned rows, and $numCols$ the number of columns of a tuple. Note that this formula needs not to consider the cost of sorting, since this is done by the sorting node itself, which is inserted if not already done.

Since the unary unification operator produces a sorted result according to all tuple attributes, this information is delivered to the planner. Using this information, the planner can optimize the operators that follow the unary unification.

In the planning stage, the unary unification statement is transformed into a planning node which is represented by the following C-struct:

```
struct UUnify
{
  Plan plan;
  int  numCols;    /* number of columns in total */
  Oid  *uniqOperators;  /* equality operators to compare with */
};
```

The first variable is of type plan struct, in which the information for the planner is stored, such as number of returned rows and cost; this variable allows the planner to process this `UUnify` struct like all other planning nodes. The second variable stores the number of columns of the input, which is needed during the execution to derive how to process the input. The last variable is an array of equality operators in order to be able to compare tuples.

Example 19. Figure 6.3 shows the graphical execution plan of the unary unification operator in our running example, using *Dept* as unification attribute. The UNION statement is executed by appending one relation to the other and then performing duplicate elimination using hash aggregation. Then the result is joined with the original relation using a hash left join, the result is sorted, and finally unary unification is applied.

6.2.5 Executor

`ExecInitUUnify`. The initialization function of the algorithm UUNIFY receives the `PlanNode` from the planner and creates a context information struct `UUnifyState`, which is initialized with the information from the planner. Then the join sub-node is initialized and stored in the variable *subnode* in the state information. Next, memory to store two input tuples, *tup* and *next*, is allocated, and a buffer, *buff*, to store two result tuples is initialized.

`ExecUUnify`. Function 6.4 shows the pseudo code of the `ExecUUnify` function, which differentiates between three types (or phases) in the tuple generation: *start*, *intermediate*, and *end*. Recall the scenario shown in Figure 6.2. The start tuple is produced over the interval from the start timestamp of r_1 to t_1. The intermediate tuple is from t_1 to t_2 (since z'_1 and z'_2 are produced by the same left tuple of the join). Finally, the end tuple is from t_2 till the end of r_1 (since the next tuple was produced from a different left tuple). If a tuple has no right part in the join, i.e., t is `NULL`, the start tuple stretches over the entire timestamp.

The function `ExecUUnify` gets as input the state information `UUnifyState` and returns either a single output tuple or `NULL`, which indicates the termination of the operation. If the function is called for the first time, it fetches two tuples from its subnode using `ExecProcNode`. If the subnode was not empty, the start tuple from the first argument tuple is generated and added to the buffer.

Next, if the buffer does not contain tuples created in this or in the previous call, the algorithm checks whether the current tuple *n.tup* is `NULL`; if so, the algorithm terminates and returns `NULL`. Otherwise, if *n.tup* and *n.next* where produced by the same tuple on the left of the join, an intermediate tuple is produced and added to the buffer; a new tuple is fetched from the subnode. If *n.tup* and *n.next* do not share the same left tuple from the join, the end tuple of the current tuple is produced, provided that the time point from the join is not `NULL`. Then, a new tuple is fetched from the subnode. Finally, if the buffer contains some tuples, the first one is retrieved and returned.

`ExecEndUUnify`. In the clean-up function of the unary unification algorithm the `ExecEnd` function of the subnode is called recursively and the memory allocated in the initialization step is released.

6.3 Binary Unification

6.3.1 Overview

The implementation of the binary unification operator in the PostgreSQL database server follows the same strategy as the implementation of unary unification. The binary unification algorithm (see Figure 4.6) contains also a nested loop which can be transformed into a join expression, which allows to take advantage of existing optimization rules and evaluation algorithms. More specifically, the binary unification algorithm is divided into the following three step:

1. perform a non-temporal Left Join to produce $\mathbf{z}'$;
2. sort $\mathbf{z}'$;

```
Algorithm: ExecUUnify
Input: State information n.
Output: A single output tuple or NULL.
if function is called for the first time then
    n.tup ← ExecProcNode(n.subnode);
    n.next ← ExecProcNode(n.subnode);
    produce start of n.tup if n.tup ≠ NULL (store in n.buff);
while n.buff is empty do
    if n.tup is NULL then
        return NULL;
    if n.tup[A] = n.next[A] ∧ n.tup.T = n.next.T then
        produce intermediate of n.tup and n.next (store in n.buff);
        n.tup ← next;
        n.next ← ExecProcNode(n.subnode);
    else
        produce end of n.tup if n.tup.t ≠ NULL (store in n.buff);
        n.tup ← next;
        n.next ← ExecProcNode(n.subnode);
        produce start of n.tup if n.tup ≠ NULL (store in n.buff);
new ← first tuple in n.buff;
remove new from n.buff;
return new;
```

Figure 6.4: Pseudo Code of `ExecUUnify`.

3. produce the result of binary unify while scanning $\mathbf{z}'$.

In step 1 a left join expression, $\mathbf{r}\Phi_\theta\mathbf{s}$ is, $\mathbf{r} \rtimes\!\!\bowtie_{\theta \wedge \mathbf{r}.T \cap \mathbf{s}.T \neq \emptyset} \mathbf{s}$, is created to produce the nested loop. Then a projection is added to remove all non-temporal attributes of $\mathbf{s}$, preserving only its timestamp attributes. Step 2 sorts the result of step 1 according to the attributes that derive from relation $\mathbf{r}$ to ensure that all tuples that are derived from the same $\mathbf{r}$ tuple are consecutive. Additionally, we also sort according to the start timestamp derived from relation $\mathbf{s}$, which corresponds to the chronological order in the inner loop of the algorithm. In step 3 the result of step 2 is scanned and the result tuples of the binary unification operator can be produced.

Figure 6.5 illustrates the binary unification using a left join. Three result tuples of step 2 are shown, where r_x and s_x correspond to the part of the join derived from the $\mathbf{r}$ and $\mathbf{s}$ relations, respectively. First, we read tuple z'_1, r_1 has no overlapping starting part with s_1, so we produce its starting not covered part. Then, we produce the intersection of r_1 and s_1 and add it to the result; we need to store how far the r_1 tuple was processed, i.e., till the end of s_1. Second, we read the second tuple z'_2. Since the tuple r_1 was not processed to the start of s_2, we need to produce this part now. Additionally, we produce the intersection part of r_1 and s_2, and store again the point up to which r_1 was processed; z'_2 becomes the previous tuple. When the third tuple z'_3 is read, we recognize that the r_x part of z'_3 is different from the r_x part of z'_2, since we always store the previous tuple to make this comparison. Therefore, the remaining part of the

previous r_x tuple can be produced, i.e., the part from the end of s_2 till the end of r_1.

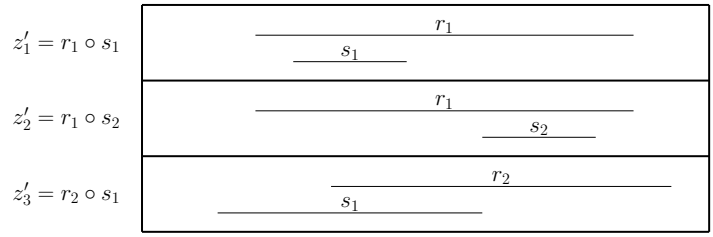


Figure 6.5: Binary Unification from Left Join.

Example 20. Consider the binary unification $\mathbf{r}\Phi_\theta\mathbf{s}$ on the running example, where $\theta \equiv \mathbf{r}.Emp = \mathbf{s}.Emp \wedge \mathbf{r}.Dept = \mathbf{s}.Dept$, which is translated into the execution of the join $\mathbf{r} \bowtie_{\theta \wedge \mathbf{r}.T \cap \mathbf{s}.T \neq \emptyset} \mathbf{s}$, followed by a projection and sorting according to the attributes $\mathbf{r}.Emp, \mathbf{r}.Dept, \mathbf{r}.T, \mathbf{s}.T$. The result of the join is shown in Table 6.3.

	Emp	*Dept*	T_r	T_s
j_1	Ann	AI	$[9, 15)$	$\perp$
j_2	Ann	DB	$[3, 8)$	$\perp$
j_3	Joe	DB	$[14, 19)$	$[12, 21)$
j_4	Sam	DB	$[1, 6)$	$[4, 11)$

Table 6.3: Join Result to Compute Binary Unification.

The binary unification algorithm processes the output of the join statement tuple by tuple. The first tuple, j_1, has a `NULL` value for the timestamp attribute, which means that the tuple has no matching tuple in $\mathbf{s}$, hence the result tuple u_1 in Table 6.4 is produced. The same holds for tuple j_2, which produces u_2. Next, the tuple j_3 is processed. The timestamp T_r is completely covered by the timestamp T_s. Thus, the operator produces just the tuple u_3 (as its intersection). When j_4 is processed, we produce the result tuple u_4 as the start part of j_4, and we know that no other tuple has common points since the relation is sorted. Then we can process the intersection of j_4 and produce tuple u_5. Since the complete tuple (Sam,DB,[1, 6)) was processed and no more tuple exist, the algorithm terminates.

u_1	Ann	AI	$[9, 15)$
u_2	Ann	DB	$[3, 8)$
u_3	Joe	DB	$[14, 19)$
u_4	Sam	DB	$[1, 4)$
u_5	Sam	DB	$[4, 6)$

Table 6.4: Result of $\mathbf{r}\Phi_\theta\mathbf{s}$.

In the following we describe the individual steps of the implementation in more detail.

6.3.2 Parser

To integrate the binary unification operator into the SQL language the keyword `UNIFY` is re-used. The syntax for the operator follows the syntax of the join, which is very similar, since both have two input relations and a condition. The required modification to the grammar of PostgreSQL's SQL is as follows:

```
unify_table:
        table_ref UNIFY table_ref join_qual
        { ... };
table_ref:
        ...
        | '(' unify_table ')' alias_clause
        { ... };
```

Two new rules are introduced into PostgreSQL's grammar file `gram.y`. The first rule specifies the operator itself. The first `table_ref` is the argument relation followed by the `UNIFY` keyword and the second `table_ref`, which is the reference relation. Finally, we have `join_qual` as join condition. The second rule integrates the new operator into the grammar by declaring it as a `table_ref`, which means the new statement can be used as an SQL `FROM` item. As an example of the new grammar, the SQL statement for the expression $\mathbf{r}\Phi_{\mathbf{r}.Emp=\mathbf{s}.Emp \wedge \mathbf{r}.Dept=\mathbf{s}.Dept}\mathbf{s}$ is as follows:

```
Select emp, dept, ts, te
From ( r Unify s
       On r.emp=s.emp And r.dept=s.dept
     ) r;
```

When the parser reduces to the unification rules, a new C-struct is created, called `BUnifyExr`, which is a new node in the parse tree of a query. The C-struct `BUnifyExr` is implemented as follows:

```
struct BUnifyExpr
{
  NodeTag   type; /* type of this struct */
  Node     *larg; /* argument subtree */
  Node     *rarg; /* reference subtree */
  Node     *quals; /* theta condition */
  ...
}
```

The first variable contains the type required by the PostgreSQL implementation to identify the type of this struct after casting. The other variables store the argument relation, the reference relation, and the θ condition.

6.3.3 Analyzer and Rewriter

The analysis phase gets the parse tree as input and transforms it to a query tree, i.e., the newly created `BUnifyExr` of the parse tree has to be analyzed and transformed to a node of the query tree structure.

As a first step the left join statement is created. That is, for the operator $\mathbf{r}\Phi_{\theta}\mathbf{s}$ a new query node $\mathbf{r} \rtimes_{\theta \wedge \mathbf{r}.T \cap \mathbf{s}.T \neq \emptyset} \mathbf{s}$ followed by a projection is created

as a subnode for the new unification statement. Then the binary unification algorithm can use the result of this statement as input. By creating this new node, the analysis for the unification (e.g., check that the relations exist, the columns are available, and the operators in the θ-condition exists) is done by the join statement. The same holds for the rewriter, since the binary unification node does not directly deal with relations, but always with its created join subnode.

6.3.4 Planner

The planning phase for the binary unification is similar as for the unary unification; if the data is not sorted, an additional sorting step is required. The approximate number of output tuples for the binary unification is computed as 3 times the number of input tuples, since an input tuple can produce at most three output tuples. The cost of performing binary unification is approximated as $cost = cpu\_operator\_cost * numRows * numCols$, similar as for unary unification. Also, the binary unification operator produces a sorted result, which is communicated to the planner.

To store all pieces of information in a plan node, the same structure as for the unary unification plan can be used, however, since we need to differentiate its type, a new C-struct is created:

```
struct BUnify
{
  Plan plan;
  int  numCols;          /* number of columns in total */
  Oid  *uniqOperators;   /* equality operators to compare with */
};
```

Example 21. Figure 6.6 shows a graphical representation of the execution plan of the binary unification operator for our running example. The left-join statement is executed using a hash-join, and the result is sorted. Then it is processed by the binary unification algorithm.

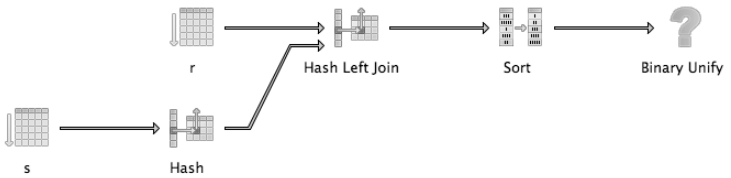


Figure 6.6: Example Explain Binary Unify.

6.3.5 Executor

In this stage the actual binary unification algorithm is implemented, which takes as input the algorithm performing the join-sub node. As described for the execution phase, 3 main functions have to be implemented in order to perform the operator. In the following the implementation of these 3 function will be described:

`ExecInitBunify`. In this initialization function, the subnode is recursively initialized, then the memory to store 2 input tuples and a buffer to store 3 output tuples is allocated. The references to the allocated memory is kept in the context information `BUnifyState`, which is returned by this function.

`ExecBUnify`. Figure 6.7 shows the pseudo code of the `ExecBUnify` function, which distinguishes between four types of tuple generation: *start*, *intersect*, *intermediate*, and *end*. Referring to Figure 6.5, the start tuple is produced as the interval from the start timestamp of r_1 to the start timestamp of s_1, which might be empty and then no tuple would be produced. The intersect tuple is produced as the intersection of the timestamps of r_1 and s_1. The intermediate tuple is produced from the end-timestamp of s_1 to the start-timestamp of $s2$, since z'_1 and z'_2 are derived from the same left-side tuple. No tuple is produced if this interval is empty. Finally, the end tuple is created from the end timestamp of s_2 till the end of r_1, since the next tuple z'_3 is derived from a different outer tuple. The end tuple is not produced when an inner tuple s_x ends after the outer r_x. If the inner (right) part of the join is empty, the entire tuple counts to the start and all others will not be produced.

`ExecBUnify` gets as input the state information `BUnifyState` and returns either a single output tuple or `NULL`, which indicates the termination of the operation. First, the function checks if it is called for the first time. If so, it fetches two tuples from its subnode using `ExecProcNode`. If the subnode was not empty, the start and intersect tuples (if any) of the first tuple are generated and added to the buffer. If the buffer does not contain tuples created in the previous execution, the function checks if the current tuple *n.tup* is `NULL`; if so, it terminates and returns `NULL`. If this is not the case, the function proceeds and checks whether the current and next tuple are produced by the same outer tuple in the join; if so, the intermediate tuple (if any) is added to *n.buff*. Then the intersect tuple of the next tuple is generated and added to the buffer, and the next tuple is fetched from the subnode. If the current and next tuples where not generated by the same outer tuple, the end tuple of *n.tup* (if any) is added to the output buffer. The next tuple is then fetched and, if the current tuple is not `NULL`, its start and intersect tuples (if any) are produced. Should the buffer contain output tuples, the first one is retrieved and returned as result.

`ExecEndBunify`. In this function the memory allocated by the `ExecInitBunify` is released and the corresponding end function for the join sub-node is called recursively to release the execution memory.

```
Algorithm: ExecBUnify
Input: State information n.
Output: A single output tuple or NULL.
if function is called for the first time then
    n.tup ← ExecProcNode(n.subnode);
    n.next ← ExecProcNode(n.subnode);
    if n.tup ≠ NULL then
        produce start of n.tup (store in n.buff);
        produce intersect of n.tup if any (store in n.buff);
while n.buff is empty do
    if n.tup is NULL then
        return NULL;
    if n.tup[A] = n.next[A] ∧ n.tup.T = n.next.T then
        produce intermediate of n.tup and n.next if any (store in n.buff);
        produce intersect of n.next (store in n.buff);
        n.tup ← next;
        n.next ← ExecProcNode(n.subnode);
    else
        produce end of n.tup if n.tup.T ≠ NULL (store in n.buff);
        n.tup ← next;
        n.next ← ExecProcNode(n.subnode);
        if n.tup ≠ NULL then
            produce start of n.tup if any (store in n.buff);
            produce intersect of n.tup if any (store in n.buff);
new ← first tuple in n.buff;
remove new from n.buff;
return new;
```

Figure 6.7: Pseudo Code of `ExecBUnify`.

Chapter 7

Evaluation

In this section we analyze the performance and scalability of our solution. First, we analyze the scalability of the unary and binary unification operator. Second, we compare our solution of reducing temporal operations to non-temporal ones using unification with the unfold mechanism that normalizes timestamps.

7.1 Setup and Data Sets

For the experiments both client and database server run on the same computer, a Mac Book Pro 2.2 GHz Core 2 Duo with 3 GB of Ram. The database server consists of a PostgreSQL server version 8.4.2, which is extended with temporal unification and the unfold mechanism. To make fair comparisons, all implementations are done directly inside the database server; no external or user-defined functions are used and no indexes on the relations are created.

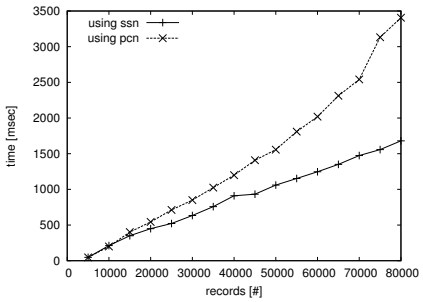
For the evaluation, the real-word data set *Incumben* of the UIS database of the University of Arizona is used. The relation has 83,857 entries, where each entry keeps track of a job assigned to an employee over a specific time interval. The data ranges over 16 years, storing the information of 49,195 different employees on a granularity of days. The minimum and maximum length of the time intervals is 1 and 573 days, respectively, and the average is approx. 180 days. To perform worst case complexity analyses, synthetic data sets were created. The properties of these data sets are described in the experiments.

7.2 Scalability of Unary and Binary Unification

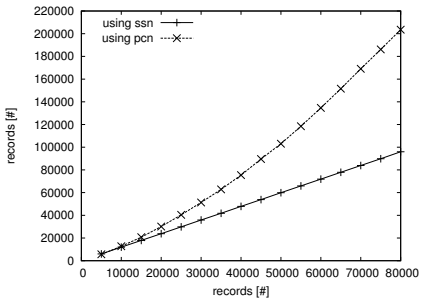
7.2.1 uUNIFY

Figure 7.1 shows the complexity of computing unary unification on the *Incumben* data set, using two different unification attributes (*ssn* and *pcn*). The graphs on the left show the runtime by varying the number of input tuples. The graphs on the right measure the number of output tuples depending on the number of input tuples.

On this real-world data set the operator shows a linear times logarithmic runtime complexity. This complexity can be explained by the underlying implementation of the operator, where the splitting points of each tuple are re-



(a) Runtime



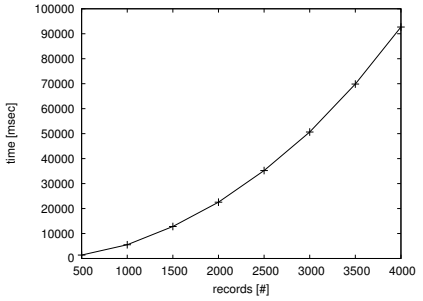
(b) Output

Figure 7.1: Unary Unification (*Incumben* Data Set).

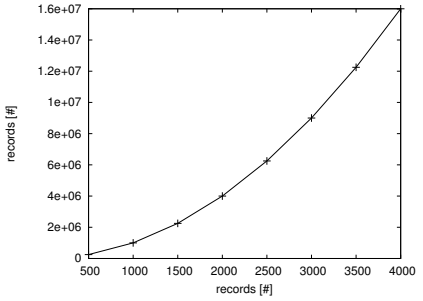
trieved using an outer join, which is the dominating factor of the algorithm. The database management system in the planning phase chooses a merge join, resulting in a linear times logarithmic complexity for this data-set.

We note a difference in the computation time and output, when different attributes are used for the unification. The unification using the *ssn* attribute is more efficient and generates less result tuples as for the *pcn* attribute. The reason for this is that the data contains less distinct values for *pcn* as for *ssn*. This results in a less efficient computation of the underlying merge join and more result tuples, due to a higher number of join matches.

To show the implications of Theorem 1 (that is, the worst case scenario for the unary unification operator), a synthetic data set *Triangle* is created following the pattern of Figure 4.2, that is, all tuples overlap with each other. For this experiment, we vary the number of input tuples, and we do not use any unification attribute, hence each tuple matches with each other. The result of this experiment is shown in Figure 7.2. Note the quadratic runtime complexity and the quadratic number of output tuples for this worst case data set, which validates Theorem 1.



(a) Runtime

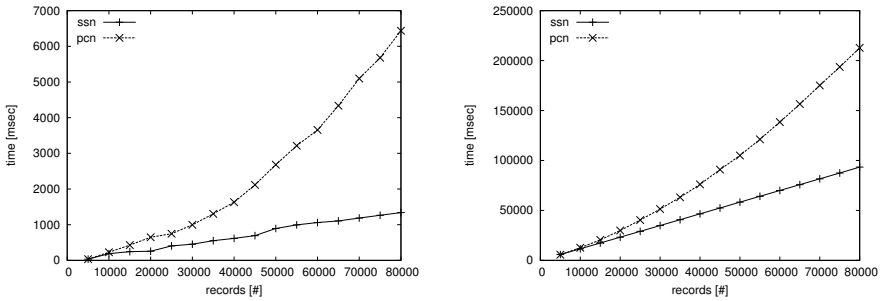


(b) Output

Figure 7.2: Unary Unification (*Triangle* Data Set).

7.2.2 bUNIFY

Figure 7.3 analyses the complexity of the binary unification on the real-world data set, using two different θ conditions. For the experiment, both the argument and the reference relation are random subsets of the same size of the *Incumben* relation. The two different θ conditions are an equality predicate over the *ssn* attribute (displayed as *ssn*) and an equality predicate over the *pcn* attribute (displayed as *pcn*), respectively.



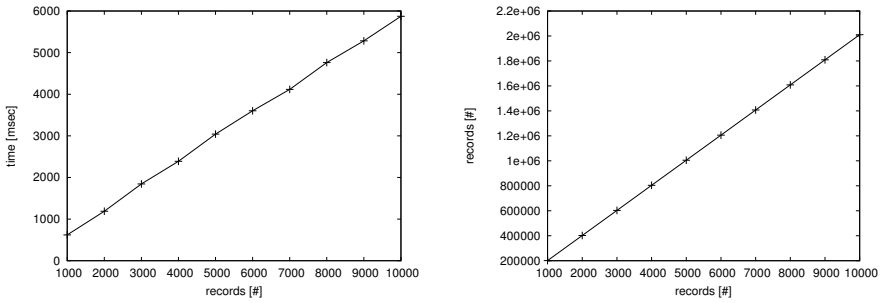
(a) Runtime (b) Output

Figure 7.3: Binary Unification (*Incumben* Data Set).

The binary unification operator with an equality predicate as θ shows a linear times logarithmic runtime behaviour. As for the unary unification, this behaviour is due to the underlying outer join in the implementation, which is performed by the database management system using a merge join. The same reasoning as for the unary unification holds for the difference between the two θ conditions. The *ssn* attribute, due to the higher number of distinct values, has a lower selectivity and is therefore more efficient for the join. The θ condition as an equality predicate over the *pcn* attribute produces more tuples, since more tuples satisfy the condition and overlap as it is the case for the *ssn*.

To show the worst case of the binary unification operator and to validate Theorem 2, a synthetic data set *Block* is created following the pattern in Figure 4.5. The cardinality of the reference relation **s** is kept constantly to 100, whereas the cardinality of the argument relation varies from 1,000 to 10,000 tuples, all ranging over the same time interval. In the operator expression the θ condition is set to the boolean predicate *true*, which means all tuple of the argument relation **r** match all tuples of the reference relation **s**.

The result of this experiment is shown in Figure 7.4. The right plot shows that the cardinality of the result is a function of the cardinalities of both relations as stated in Theorem 2; the graph shows a linear behaviour since the cardinality of the reference relation is fixed. The same holds for the runtime; the cardinality of the reference relation is fixed to 100, resulting in a linear curve. Notice that the logarithmic part contributed by the sorting step is not visible, since the data sets are relatively small and the sorting is mostly done in memory.



(a) Runtime (b) Output

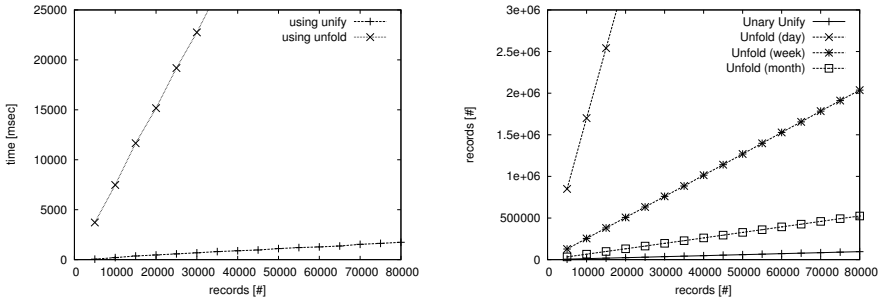
Figure 7.4: Binary Unification (*Block* Data Set).

7.3 Temporal Operators

In this section we compare our solution of reducing temporal operators to non-temporal operators by using temporal unification with a similar reduction using the unfold mechanism for timestamp normalization proposed in IXSQL [9, 14]. For this the *unfold* operator has been implemented in PostgreSQL in order to transform a relation from an interval based temporal relation into a point based temporal relation, where a single tuple stores exactly one time point. The evaluation of the reduction is done for three temporal operators, namely aggregation, difference, and join.

7.3.1 Aggregation

Figure 7.5(a) compares the runtime of computing the temporal aggregation ${}_{ssn}\vartheta^{T}_{Count(*)}(Incumben)$ for the two different ways of reduction. The reduction using unary unification clearly outperforms the computation of temporal aggregation using the unfold operator. Both show a linear times logarithmic complexity due to the required sorting: for the unary unification the sorting is required for the outer join for which a merge join is used; in the case of the unfold the non-temporal aggregation is done by sorting.



(a) Aggregation (b) Intermediate Results

Figure 7.5: Aggregation (*Incumben* Data Set).

The explanation for this huge difference in the runtime is shown in Fig-

ure 7.5(b), which compares the output of the unfold operator to the output of unary unification. The non-temporal aggregation is actually applied to this intermediate result to produce the final result. With a granularity of days, at which the *Incumben* data set is stored, the unfold operator produces on average 180 tuples for each input tuple, which corresponds to the average length of the time-intervals. The unary unification operator instead produces far less output tuples, giving a much smaller intermediate result as input for the non-temporal aggregation operator. The plot shows two more unfold operations, using a granularity of weeks and months, respectively. A larger granularity reduces the cost of aggregation, however, they are not applicable for this data set without loss of information.

7.3.2 Difference

The runtime behaviour of the temporal difference is shown in Figure 7.6. The computation of the temporal difference using binary unification is orders of magnitudes faster than using the unfold operator. The large intermediate result of the unfold operator compared to binary unification has a notable effect on the runtime of the non-temporal difference.

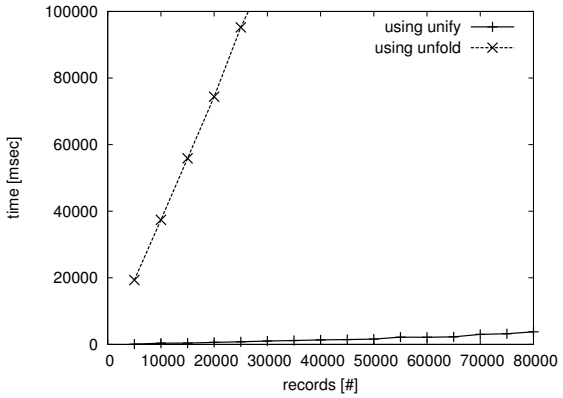


Figure 7.6: Difference (*Incumben* Data Set).

For both solutions the runtime shows a linear times logarithmic behaviour: for the unary unification due to the outer join that is executed as a merge join, and for the case of unfold due to the sorting step done by the database management system to produce the non-temporal difference. Also for the difference a larger granularity would be favorable for the unfold, whereas our solution with unification is granularity independent.

7.3.3 Join

The runtime behaviour of a temporal equi-join over the *ssn* attribute of the *Incumben* data set is shown in Figure 7.7. As it was the case for temporal difference, also in this case the reduction using binary unification performs orders of magnitudes better as the computation of the temporal join using unfold. Although both operations have linear times logarithmic running time behaviour

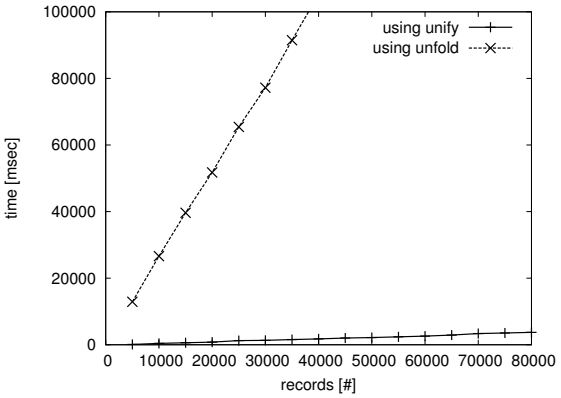


Figure 7.7: Join (*Incumben* Data Set).

due to the sorting, the running time of the non-temporal join suffers from the large intermediate result produced by the unfold operator, whereas the intermediate result of binary unification is much smaller, leading to a more efficient join execution.

7.3.4 Summary of the Evaluation

The empirical evaluation shows that temporal unification provides a scalable solution for the reduction of temporal operations to non-temporal operations. For all cases, temporal unification clearly outperform the unfold operator by orders of magnitudes. The main reason for this is the smaller intermediate result that is generated by unification compared to unfold, which allows a more efficient execution of the subsequent non-temporal operators.

The main factors affecting the performance of the unary unification operator are the choice of the unification attributes, and the frequency of overlapping intervals in the data. When many tuples in the data share the same values for the unification attributes, the operator tends towards quadratic time complexity, due to the underlying join. A high frequency of overlapping timestamps in the data is only an issue if it occurs in combination with the first factor, i.e., a high number of equal unification attribute values. In this case the operator approaches is worst case and it tends to produce quadratic output. However, such data shall be very rare in real-world applications.

Also in the case of reducing binary operations, binary unification scales much better as the unfold operator. Like for unary unification, binary unification produces less tuples than unfold, which makes the non-temporal operators more efficient. The factors affecting the performance of binary unification are the θ-condition and the number of overlapping input tuples. The θ-condition affects the run-time behaviour of the underlying join, whereas the number of matching and overlapping tuples affects its output complexity.

Chapter 8

Conclusion and Future Work

In this thesis we provide a novel solution to provide support for managing temporal data in relational database systems in a principled way. Our solution is based on two new operators, termed unary and binary temporal unification, which allow to reduce a temporal relational algebra to the non-temporal relational algebra. Using these unification operators, we provide reduction rules for the most important temporal operators. Our solution preserves lineage information and takes advantage of existing database technologies for non-temporal data.

For the computation of the unary and binary unification operators, two algorithms are provided. The operators are implemented in the core of the PostgreSQL database system, by first extending the SQL language, then modifying the parser and analyzer of PostgreSQL, and finally integrating the provided algorithms into the execution core of the database management system. The implementation ensures to minimize the overhead for input and output compared to traditional middle-ware solutions.

In extensive experiments we analyze the scalability of our solution and compare its performance to an approach that is based on timestamp normalization as proposed in IXSQL. The experiments show that the unification operators clearly outperform the unfold approach by orders of magnitudes. For operations such as aggregation, difference, and equi-join, the operators show a linear times logarithmic runtime behaviour. By using synthetic datasets, we have shown the worst case scenario with a quadratic runtime complexity, though such data-sets are very rare in real-world applications.

Future work includes the following aspects. First, we will investigate how to improve the outer joins in the implementation of the unification operators by using some advanced indexing technique (for the case that no conventional join technique can be evaluated efficiently). Second, we will study more accurate cost estimations in order to improve the optimizer. Third, we will extend the temporal unification operators to support also temporal bags.

Bibliography

[1] M. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT-2006)*, number 3896 in LNCS, pages 257–275, Munich, Germany, Mar. 2006. Springer Verlag.

[2] M. H. Böhlen, J. Gamper, C. S. Jensen, and R. T. Snodgrass. SQL-based temporal query languages. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2762–2768. Springer Verlag, 2009.

[3] M. H. Böhlen and C. S. Jensen. *Encyclopedia of Information Systems*, chapter Temporal Data Model and Query Language Concepts. Academic Press, 2002.

[4] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating the completeness of TSQL2. In *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*, pages 153–172, Zürich, Switzerland, September 1995. Springer, Berlin.

[5] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):48, December 2000.

[6] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in temporal databases. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers, Mumbai (Bombay), India, September 1996.

[7] J. Gamper, M. H. Böhlen, and C. S. Jensen. Temporal aggregation. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2924–2929. Springer Verlag, 2009.

[8] D. Gao, S. Jensen, T. Snodgrass, and D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, 2005.

[9] H. D. J. Date and N. Lorentzos. *Temporal Data and the Relational Model*. Morgen Kaufmann Publisher, 2002.

[10] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying temporal models via a conceptual model. *Information Systems*, 19(7):513–547, 1994.

[11] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proceedings of 11th International Conference on Data Engineering (ICDE-95)*, pages 222–231, Taipei, Taiwan, Mar. 1995.

[12] N. Kline, R. T. Snodgrass, and T. Y. C. Leung. Aggregates. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 21, pages 395–425. Kluwer Academic Publishers, 1995.

[13] N. Lorentzos and R. Johnson. Extending relational algebra to manipulate temporal data. *Information Systems*, 15(3), 1988.

[14] N. A. Lorentzos and Y. G. Mitsopoulos. SQL extension for interval data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, May/June 1997.

[15] B. Moon, I. F. Vega Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):744–759, 2003.

[16] C. Murray. Oracle database workspace manager developer's guide. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28396.pdf, 2008.

[17] PostgreSQL. Online temporal PostgreSQL reference. http://temporal.projects.postgresql.org/reference.html.

[18] A. Segev. Join Processing and Optimization in Temporal Relational Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 15, pages 356–387. Benjamin/Cummings Publishing Company, 1993.

[19] R. T. Snodgrass. *Developing Time-Oriented Database Application in SQL.* Morgen Kaufmann Publisher, 1999.

[20] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings of the International Conference on Data Engineering*, pages 282–292, February 1994.

[21] Teradata. Teradata database temporal table support. www.info.teradata.com/edownload.cfm?itemid=102320064, 2010.

[22] D. Toman. Point-based vs interval-based temporal query languages. In *Proceedings of the* 15^{th} *ACM Symposium on Principles of Database Systems*, pages 58–67, Montreal, Canada, June 1996.

[23] D. Toman. Point-based temporal extensions of SQL. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, 1997.

[24] D. Toman. Point-based temporal extensions of SQL and their efficient implementation. In O. Etzion, S. Jajodia, and S. M. Sripada, editors, *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 211–237. Springer, 1998.

[25] I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 17(2):271–286, 2005.

[26] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB Journal*, 12(3):262–283, 2003.

[27] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *Proceedings of the 18th International Conference on Data Engineering (ICDE-02)*, pages 103–113, San Jose, 2002.