Support for Temporal Data by Complex Objects

W. Käfer, N. Ritter, H. Schöning

University Kaiserslautern, Department of Computer Science,
P.O. Box 3049, D-6750 Kaiserslautern, West Germany
email: kaefer@informatik.uni-kl.de

Abstract

Support for temporal data continues to be a requirement posed by many applications. We show that a complex object data model is an appropriate means for handling temporal data. Firstly, we describe the main features of temporal databases in terms of time sequences, valid time, etc. We then explain the mapping of time sequences onto recursively structured complex objects. Operations on temporal data are easily transformed into complex object operations. To cope with the huge storage requirements arising from temporal databases, we integrate the concept of storing logical differences into our approach. Here, we exploit the extensibility of the underlying complex object's database system PRIMA. Finally, we briefly sketch a further improvement to guarantee fast access to the present data by storing them apart from the historical data without loosing the connection between both.

1. Introduction

All human activities are embedded in time, but conventional database systems do not possess the capability to record and process the dynamic aspects of the changing world. The need to support the time dimension in database systems is obvious in applications like banking, sales, etc. Even most of the well-known employee database models neglect the fact that the history of an employee (at least the episode in which he or she was with the enterprise) is urgently needed for management tasks.

There is plenty of literature related to the concept of time, particularly in the area of the relational model [Bo82, CW83, Ga88, SK86, Sn86, Ta86]. *Temporal databases* (as defined by [AS85, AS86a]) capture the history of retroactive and prospective changes and allow for the derivation of facts from the database because of their temporal interdependence. Most of the prototype implementations of temporal databases rely on the relational model (such as [Sn87] based on Quel and [Ar86] based on SQL). However, the modeling of temporal data with the relational model has some serious drawbacks. The existence of flat relations means that the history of a single entity has to be smashed into many pieces: each tuple represents a snapshot of an entity at a certain time. There are many snapshots of the same entity, requiring that the primary key has to be expanded by the time of the snapshot. There is no notion to capture the complete history of an entity as a whole. Therefore, some queries become more complicated, because they have to consider the distribution of one entity over many tuples.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

We want to overcome these problems by the use of a data model which supports complex objects. The history of an entity can be modeled as one complex object. Thus, we can treat the complete history of an entity as one unit. Furthermore, we can use the powerful query language of the complex object data model to perform the selection of histories or parts of them. As an example, to determine the salary of the employee Mary at a single point in time, we have to perform a query, which

- selects the complex object (representing the history of the employee) and
- selects the salary valid at the appropriate time.

Using a complex object database system as the basis for the mapping process from temporal data and temporal queries to complex objects and queries on them provides us with many advantages:

- The implementation of the temporal database system should be easy, using an enhanced database system.
- Since complex object queries are executed efficiently, the corresponding temporal data should be handled efficiently as well.
- Creating and removing access path structures on the temporal data is very flexible because of the underlying enhanced database system.
- Handling of temporal and non-temporal data can be done in a uniform way.

Figure 1.1 illustrates our approach.

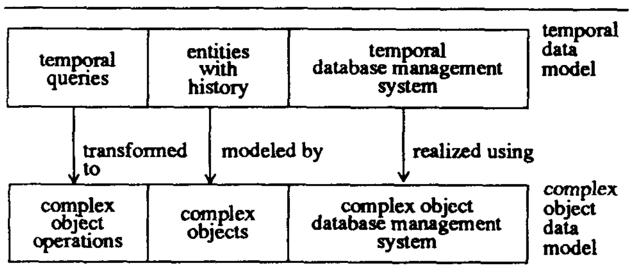

temporal queries
entities with history
temporal database management system
temporal data model
transformed to
modeled by
realized using
complex object operations
complex objects
complex object database management system
complex object data model

Figure 1.1: The mapping of the temporal data model to the complex object data model

The rest of the paper is structured as follows: Section 2 introduces the major features of our temporal data model. Based on these features, we briefly discuss some advantages of mapping this temporal data model onto a complex object data model. We then identify the MAD model [Mi88, Mi89] as a well-suited candidate for a target data model. Section 3 describes the features of the MAD model which we will use in our approach, and the mapping of temporal objects to complex objects. The transformation of temporal queries to MAD operations is explained in section 4. Furthermore, we employ the concept of reverse differences to reduce the storage space for temporal objects. To improve the access time to the latest data (which are expected to be accessed most frequently), we separate them from historical data. Section 5 contains a conclusion and an outlook as to further work.

2. The Temporal Data Model

In this section, we sketch some basic features of our temporal data model which are necessary to understand the mapping process to the complex object data model. Due to space limitations, we cannot, however, demonstrate the whole functionality of our model. An in-depth discussion of the temporal data model is found in [Kä90]. Its functionality is similar to that of other temporal database systems, such as described in [Ar86, Sn87, SK86]. The temporal data model we will discuss in the following is also based on, but not limited by the relational model.

Conventional relational databases represent the state of an application such as the staff management of an enterprise at a single moment in time. Roughly speaking, each tuple in the database represents a snapshot of a real world entity, e.g. of the employee Mary in March 1980. Each change in Mary's data is represented by a change of the corresponding tuple, updating the snapshot and overwriting the previous values. Thus, an earlier state of Mary's history cannot be retrieved from the database. In the temporal data model, we preserve all these snapshots (tuples) in a time-ordered sequence, the so-called *Time Sequence* (TS) [SK86]. A TS represents the history of an entity of the real world and can be seen as an extension of a tuple in the temporal dimension, i.e., each TS

- belongs to exactly one TS relation (like a tuple belongs to one relation),
- has a unique key (surrogate) and
- serves as a unit for retrieval and manipulation operations.

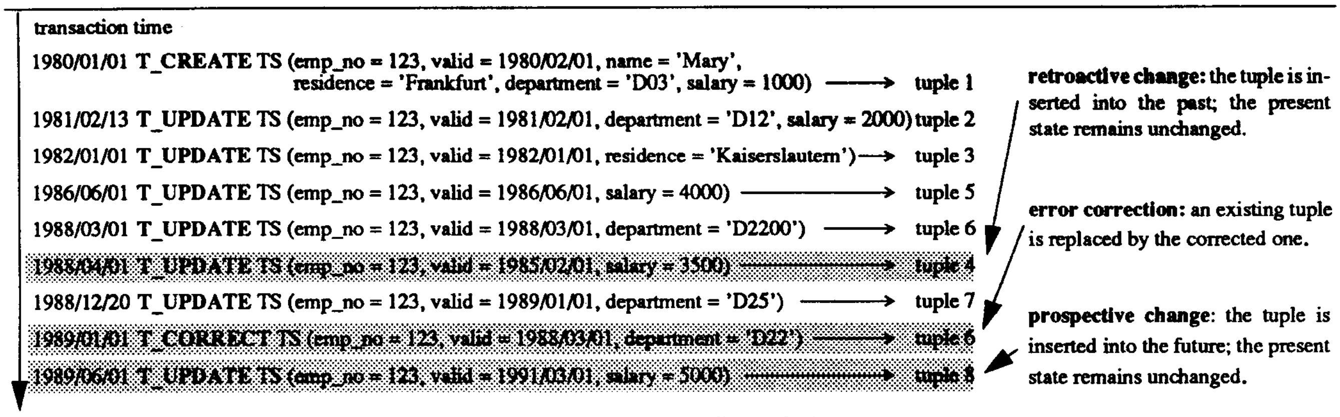
Figure 2.1 shows the TS representing an episode of Mary's life. She joined the enterprise at 1980/02/01. At this time she lived in Frankfurt and was associated with department D03 (*tuple 1*). After one year she joined department D12 and was earning a salary of $2000 (*tuple 2*). After several changes of her residence, department and her salary (*tuple 3* through *tuple 6*), she is now assigned to department D25, lives in Kaiserslautern and earns a salary of $4000 (*tuple 7*). In March 1991 an increase of salary is proposed leading to a tuple (*tuple 8*) which belongs to the future.

In our example, the history of Mary is stepwise constant, i.e. each value of an attribute is valid until it is changed. Therefore, we can determine the value of an attribute at each time (within the lifetime of the entity) by looking at the tuple in the TS with the latest time which is less or equal to the requested time. Besides this kind of history, TS are able to represent event-oriented and continuous history [Kl81, SK86]. In event-oriented histories (e.g. debit/credit actions on an account) the tuples in the TS are only valid at single points in time. In the case of continuous histories, such as a fever

time ↑	valid	emp_no	name	residence	department	salary	timestamp
tuple 8	1991/03/01	123	Mary	Kaiserslautern	D25	▨ 5000	1989/06/01
tuple 7	1989/01/01	123	Mary	Kaiserslautern	▨ D25	4000	1988/12/20
tuple 6	1988/03/01	123	Mary	Kaiserslautern	▨ D22	4000	1989/01/01
tuple 5	1986/06/01	123	Mary	Kaiserslautern	D12	▨ 4000	1986/06/01
tuple 4	1985/02/01	123	Mary	Kaiserslautern	D12	▨ 3500	1988/04/01
tuple 3	1982/01/01	123	Mary	▨ Kaiserslautern	D12	2000	1982/01/01
tuple 2	1981/02/01	123	Mary	Frankfurt	▨ D12	▨ 2000	1981/02/13
tuple 1	1980/02/01	123	Mary	Frankfurt	D03	1000	1980/01/01

▨ changes;
"timestamp" represents the time, when the database has been modified.

Figure 2.1: History of employee Mary


transaction time
retroactive change: the tuple is inserted into the past; the present state remains unchanged.
error correction: an existing tuple is replaced by the corrected one.
prospective change: the tuple is inserted into the future; the present state remains unchanged.

```
1980/01/01 T_CREATE TS (emp_no = 123, valid = 1980/02/01, name = 'Mary',
                  residence = 'Frankfurt', department = 'D03', salary = 1000) ——→ tuple 1
1981/02/13 T_UPDATE TS (emp_no = 123, valid = 1981/02/01, department = 'D12', salary = 2000) tuple 2
1982/01/01 T_UPDATE TS (emp_no = 123, valid = 1982/01/01, residence = 'Kaiserslautern')—→ tuple 3
1986/06/01 T_UPDATE TS (emp_no = 123, valid = 1986/06/01, salary = 4000) ——→ tuple 5
1988/03/01 T_UPDATE TS (emp_no = 123, valid = 1988/03/01, department = 'D2200') ——→ tuple 6
1988/04/01 T_UPDATE TS (emp_no = 123, valid = 1985/02/01, salary = 3500) ——→ tuple 4
1988/12/20 T_UPDATE TS (emp_no = 123, valid = 1989/01/01, department = 'D25') ——→ tuple 7
1989/01/01 T_CORRECT TS (emp_no = 123, valid = 1988/03/01, department = 'D22') ——→ tuple 6
1989/06/01 T_UPDATE TS (emp_no = 123, valid = 1991/03/01, salary = 5000) ——→ tuple 8
```

Figure 2.2: Sequence of operations to construct Mary's history (see figure 2.1)

curve of a patient, the degree of the fever changes continuously over time. Therefore the tuples in the TS can only capture some characteristic values at single moments. Based on these characteristic values, the history has to be reconstructed by a function, e.g. linear interpolation

.In order to support audit requirements and to keep track of corrections, we have to differentiate between the time when a value is valid in the real world (*valid* time) and the time when the associated transaction runs on the database (*transaction* time). Since the transaction time is often only used for bookkeeping purposes, we order the tuples in a TS by their valid time (as shown in fig. 2.1). Consequently, we have to admit tuples with valid times belonging to the future.

Besides the notion of TS, we need corresponding operations to reflect the evolution of the TS. Figure 2.2 shows a sequence of such operations to construct Mary's history. As mentioned in the introduction, we have to consider retroactive (*tuple 4*) and prospective (*tuple 8*) changes, i.e. changes corresponding to valid times belonging to the past or to the future. Furthermore, we must be able to handle corrections (*tuple 6*). The following operations are supported to perform changes on TS[†]:

T_CREATE TS (*valid time, data*):
Similar to the well-known APPEND or INSERT operations in relational database systems, **T_CREATE TS** generates a TS with one tuple. *valid time* is the birth date of the TS.

T_UPDATE TS (*valid time, data*):
As in the case of relational systems, **T_UPDATE** describes the changes which happen to the entity, but contrary to these systems we do not overwrite the previous data. Depending on the valid time, a new tuple is generated and appended at the end of a TS (the "usual" case) or inserted into the TS. Of course, a tuple with the same valid time must not be present in the TS.

[†] In order to distinguish the temporal operations from the other operations, we precede their names with T_.

T_CORRECT TS (*valid time, data*):
The tuple with the specified valid time is replaced by the new one (error correction)[†]

T_DELETE TS (*valid time*):
A "gravestone" for the TS is generated, i.e., no further updates are allowed on the TS (except retroactive corrections).

T_REMOVE TS:
All data of the TS are discarded (this operation serves only for administrative purposes).

All these operations demand the specification of the TS relation(s) (**T_FROM** clause) and the specification of the TS itself (**T_WHERE** clause). Our language is similar to the well-known SQL skeleton. We will discuss this in more detail for the case of the **T_SELECT** operation:

T_SELECT	projection clause
	(time projection clause)
T_FROM	TS relation(s)
T_WHERE	restriction clause
	(time restriction clause);

The **T_FROM** clause enumerates the TS relations which are relevant for the statement. The expressions in the **T_WHERE** clause restrict the TS belonging to the result of the query. Thus, the result of evaluating the **T_FROM** and the **T_WHERE** clause is a set of TS belonging to one TS relation. In addition to the restriction clause known from SQL, we offer the following basic predicates to describe the time relation of the restriction clause.

2.1: juncture query

(*bool*) AT (*t*):
if the boolean expression *bool* can be evaluated to **TRUE** at time *t*, the corresponding TS belongs to the result.

[†] The old tuple is not deleted, but is used to keep a history of correction on the TS.

Example: "Retrieve all Employees who worked for department D12 at June 1st, 1981."

```
T_SELECT   ALL
T_FROM     Employee
T_WHERE    (department = 'D12') AT 1981/06/01;
```

2.2: existential interval query

(*bool*) **SOMETIMES DURING** [*t1*, *t2*]:
A TS qualifies, if the boolean expression *bool* is evaluated to be **TRUE** at least at one moment in time within the interval from *t1* to *t2*.

Example: "Retrieve all employees who worked for department D03 in 1980."

```
T_SELECT   ALL
T_FROM     Employee
T_WHERE    (department = 'D03') SOMETIMES
           DURING [1980/01/01, 1980/12/31];
```

2.3: universal interval query

(*bool*) **ALWAYS DURING** [*t1*, *t2*]:
A TS qualifies, if the boolean expression *bool* is evaluated to be **TRUE** within the whole interval.

Example: "Retrieve all employees who worked for D03 during the whole year 1980."

```
T_SELECT   ALL
T_FROM     Employee
T_WHERE    (department = 'D03') ALWAYS
           DURING [1980/01/01, 1980/12/31];
```

2.4: existential coincidence query

(*bool1*) **SOMETIMES DURING WHILE** (*bool2*):
A TS qualifies, if *bool1* holds at least at one moment in time when *bool2* has been **TRUE**.

Example: "Retrieve all Employees who earned more than $3000 in department D12."

```
T_SELECT   ALL
T_FROM     Employee
T_WHERE    (department = 'D12' ) SOMETIMES
           DURING WHILE (salary > 3000);
```

2.5: universal coincidence query

(*bool1*) **ALWAYS DURING WHILE** (*bool2*):
A TS qualifies, if *bool1* holds whenever *bool2* has been **TRUE**.

Example: "Retrieve all Employees who were assigned to department D12 whenever they earned more than $3000."

```
T_SELECT   ALL
T_FROM     Employee
T_WHERE    (department = 'D12' ) ALWAYS
           DURING WHILE (salary > 3000);
```

The employee Mary of our example qualifies with respect to queries 2.1, 2.2, and 2.4. As mentioned above, the result of each of these queries is a set of TS belonging to one TS relation (i.e. in the case of Mary the TS consists of the whole information as illustrated in Fig. 2.1). We extend the projection clause in an analogous fashion to the extensions of the restriction clause's facilities with respect to the temporal dimension. We allow for a projection of a set of slices of an entity's history. In examples above, we would get the complete history (the whole TS) of the employees due to the keyword **ALL**. In order to be more specific, we must be able to restrict the tuples of the TS contained in the result set according to certain criteria. For example, adding the **ONLY** keyword to the **T_SELECT** clause of query 2.1 would reduce the result set to the tuple valid at June 1st, 1981 of the qualifying TS (i.e. *tuple* 2).

Besides the **ONLY** clause, there are three other clauses used to restrict the TS of the result to interesting tuples. **AT** and **DURING** work analogously to the predicates in the **T_WHERE** clause. **ALLTIME** does not perform any selection on the query's result and serves as default.

time_projection ::=

ALLTIME/	: selects the whole TS.
ONLY/	: selects only the tuples qualified by the restriction clause.
AT (*t*)/	: selects the tuple valid at *t*.
DURING [*t1*,*t2*]	: selects all tuples valid in the interval from *t1* to *t2*.

The combination of the introduced predicates leads to a powerful query language. Due to space limitations, we omit further examples and an in-depth discussion of our temporal query language.

A TS as described above consists of several tuples. In a stepwise constant history, each of these tuples represents an interval in the history of the corresponding entity, during which the represented values did not change. Obviously, not only the values of the tuples are carrying information, but also their succession. For example, to decide how long the values contained in one tuple have been valid, one has to look at the successor tuple. This gives a hint to the difficulties of mapping one TS to more than one object of a data model, for example, to map a TS to a set of tuples in the relational model. Whereas simple operations can be transformed to operations of the data model (using constructs like GROUP_BY to formulate the coherence of the tuples of one TS), this is much more difficult for operations which perform computations using the temporal order existing on the tuples. Therefore, we decided to investigate the mapping of the temporal data model to a complex object model, which allows us to model one TS as one complex object, thereby preserving the coherency of the tuples. The complex object model has to fulfil the following requirements to be well-suited for this purpose:

- It must be possible to have an arbitrary number of tuples for a TS.
- It must be possible to express the temporal order inherent in a TS. This order is defined on the valid attribute of a TS.
- It must be possible to relate the values contained in subsequent time tuples to one another.
- The mapping process is simpler, if a temporal data type with corresponding operations is supported.
- The result of queries on TS should reflect the order of the tuples, i.e. should not consist of an unstructured set of unrelated values.

Regarding the NF^2 data model [SS86] as a prominent example, we can model a TS as a tuple in a relation which has a sub-relation "TS_tuples" containing the tuples representing the TS. In the pure NF^2 model, however, there is no way to arrange the tuples in a certain order, since relations are unordered by definition. The extended NF^2 model [Da86] supports lists of tuples which can be used for this purpose. A temporal data type is contained in neither of these NF^2 models.

The MAD model [Mi88, Mi89] used in this paper to illustrate the mapping process offers another way to express the coherency relation among tuples. Since it supports recursively structured complex objects, one can model a TS as an arbitrary length chain of temporal tuples. Succeeding levels of recursion (corresponding to succeeding positions in a list) represent succeeding tuples. Furthermore, MAD also has a specific temporal data type. Thus, it seems to be a well-suited data model for our investigations. Nevertheless, we do not claim that it is the only one.

3. Mapping Time Sequences To Molecules of the MAD-Model

Atoms are the basic building blocks of the MAD model. They can be compared to relational tuples in the relational model in that they consist of attributes of various types and belong to exactly one atom type (comparable to a relation).

Besides the data types known from many implementations of the relational model, some additional data types may be chosen as ranges for attributes:

- The data type TIME can be used to represent time with varying precisions. A value of this type is represented in the form **year/month/day/hour/minute/second/millisecond**, where the components can be left out from right to left. For example, 1989/02/01 is a correctly formed value of data type TIME, representing February 1st, 1989. It is said to be of granule DAY. The granule of a TIME attribute is defined in the database schema.
- For this data type, the usual comparison operators are defined. Furthermore, there is a time-oriented arithmetic, which can be illustrated by the difference between the notation one month (0/1) and 30 days (0/0/30):
  1989/04/01 + 0/0/30 = 1989/05/01 1989/04/01 + 0/1 = 1989/05/01
  1989/02/01 + 0/0/30 = 1989/03/03 1989/02/01 + 0/1 = 1989/03/01
- The data type IDENTIFIER represents a system defined surrogate which uniquely identifies an atom. Each atom type must contain exactly one attribute of this type.
- The data type REFERENCE is needed to link atoms together: A value of type REFERENCE is a duplicate-free list of IDENTIFIER values, all pointing to atoms of the same type (cf. Figure 3.1).

TS relation definitions can be mapped onto MAD atom type definitions in a quite straight-forward way. Figure 3.1 shows a schema definition of the employee TS relation and its corresponding definition as an atom type with recursive self-references in the MAD model. Besides the attributes defined in the employee TS relation, some system defined attributes are added in the MAD atom type, which are of course not visible to the user of the temporal data model. The attribute **alive** represents the gravestone: if **alive** is **FALSE** then **valid** contains the death date of the TS. The attribute **timestamp** contains the transaction time, i.e., the time when the transaction modified the tuple. The attributes **future** and **past** are used to establish a link type which chains together all atoms belonging to the same TS. Notice that **valid** is not a system-defined attribute, since it is visible to the user of the temporal data model. Furthermore, the granule of **valid** must be specified by the user.

Some characteristics of the MAD model should be stressed regarding this example (cf. Figure 3.1): for each REFERENCE attribute, there is a corresponding "counter reference" attribute pointing to the opposite direction (here: **past, future**).

The concept of REFERENCE attributes allows for the direct mapping of attribute-free binary relationships, even in the m:n case. There may be cardinality restrictions indicating the permissible number of IDENTIFIER values for a REFERENCE attribute. Thus, the [0,1] cardinality restriction of the example indicates that there may be at most one successor and predecessor in the temporal dimension.

The REFERENCE attributes may be used to dynamically define complex object structures, called molecules. For example, the molecule type definition

E1(Employee_tuple).past—E2(Employee_tuple)

TS definition		*corresponding MAD atom type definition*	
T_CREATE TS Employee		**CREATE ATOM_TYPE** Employee_tuple	
		(Id	: IDENTIFIER;
(valid	: TIME (DAY);	valid	: TIME (DAY);
emp_no	: INTEGER;	emp_no	: INTEGER;
name	: STRING;	name	: STRING;
residence	: STRING;	residence	: STRING;
department	: STRING;	department	: STRING;
salary	: INTEGER);	salary	: INTEGER;
		future	: REFERENCE (Employee_tuple.past) [0,1];
		past	: REFERENCE (Employee_tuple.future) [0,1];
		alive	: BOOLEAN;
system defined attributes		timestamp	: TIME (MILLISEC));

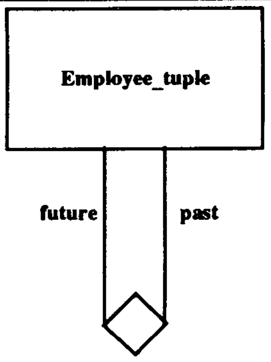


Figure 3.1: Mapping of TS definition to MAD definition

constructs molecules which cluster pairs of Employee_tuples together which are adjacent in time. When an atom type is included in a molecule type definition more than once, each of its occurrences (called a role) must be named differently (E1, E2 in the example above).

Based on the molecule type definition facility, queries can be formulated using the molecule query language (MQL) of the MAD-Model, which is based on a SELECT-FROM-WHERE skeleton similar to SQL:

```
3.1: SELECT   projection clause
     FROM     molecule type definition(s)
     WHERE    restriction clause;
```

For example, the following query selects all salary changes of Employees (at any time):

```
3.2: SELECT   ALL
     FROM     E1(Employee_tuple).future
              —E2(Employee_tuple)
     WHERE    E1.salary <> E2.salary;
```

Notice, that the atom type structure does not imply the direction imposed by the molecule type definition. In the above query, we do one step into the future from the E1 to find its successor in time, while in the query before, we went one step into the past finding the predecessor of E1.

The resulting molecules of a query may be tailored to the needs of an application, by specifying attributes and atom types to be projected in the projection clause (instead of ALL in the above example). Furthermore, a qualified (i.e. value-dependent) projection is allowed as illustrated in the following query. Suppose you want to retrieve the old salary of the employees in the above example, but retrieve the new one only if it is less than $1500. In this case you can ask the following query:

```
3.3: SELECT   E1.salary, ( SELECT E2.salary
                           FROM   RESULT
                           WHERE  E2.salary < 1500)
     FROM     E1(Employee_tuple).future
              —E2(Employee_tuple)
     WHERE    E1.salary <> E2.salary;
```

The keyword **RESULT** indicates that the **SELECT** query refers to parts of the result obtained by the surrounding query. As already mentioned, TS will be modeled as recursively structured complex objects (molecules). MAD offers the keyword **REC_PATH** for the construction of recursive complex objects. The keyword **UNTIL** can be used to cut the construction of the transitive closure when a certain condition is fulfilled. For example, the following query delivers the history of all D22 employees since they joined the department:

```
3.4: SELECT   ALL
     FROM     Employee_tuple REC_PATH
              Employee_tuple.past—Employee_tuple
              UNTIL
              Employee_tuple.department <> 'D22'
     WHERE    Employee_tuple(FIRST).department
              = 'D22';
```

The algorithm for computing the corresponding molecule consists of starting with an *Employee* fulfilling the condition *Employee_tuple(FIRST).department = 'D22'*. *Employee_tuple(FIRST)* is the name of the first *Employee* atom in the molecule (i.e., the root of the molecule). Then, the Employee_tuple referenced by *the past attribute* of the root is included into the molecule. Its **past** attribute is used to find the next level of recursion, and so on, until the **UNTIL** clause is evaluated to be **TRUE** or the **past** attribute is empty[†]. Thus, recursion terminates in our example, whenever a *department other than D22* is considered.

The keywords **PREVIOUS** and **NEXT** allow for path-dependent recursion termination. Thus, the following query delivers a set of histories for Mary, one history molecule for each department she has worked for:

[†]In our application, the atom network established by the past-future references of Employee_tuple is cycle-free. In the case of cyclic data, recursion terminates whenever a cycle would appear in the molecule.

```
3.5: SELECT ALL
     FROM   Employee_tuple REC_PATH
            Employee_tuple.past—Employee_tuple
            UNTIL
            Employee_tuple(PREVIOUS).department
            <> Employee_tuple(NEXT).department
     WHERE  Employee_tuple(FIRST).name = 'Mary' ;
```

A more general description of the MAD model can be found in [Mi88], a detailed discussion of recursion in the MAD model is given in [Schö89].

Join is not a frequently used operation in MAD, because in most cases references (i.e. user defined relationships) will be used to combine atom types (where in the relational model joins based on primary keys and foreign keys would be used). Nevertheless, molecule join is possible in MAD in analogy to SQL by enumeration of the corresponding molecule types in the **FROM** clause. To facilitate the use of frequently needed molecule type definitions, molecule types can be defined in a similar way to macro definitions with parameters†. The following example defines a molecule type for the representation of the employee time sequences. The predicate *Employee_tuple(FIRST).future = EMPTY* forces the molecules to start at the beginning of the time sequence. The **UNTIL** clause cuts the molecules at a certain point in time, i.e. tuples representing older events are excluded from the molecule. In this case, the last tuple of this molecule keeps the information valid at %DATE (which is a parameter of the macro definition). The molecule definition also covers molecules of employees, which were not alive at %DATE. In this case, molecule construction is terminated by an empty **past** reference.

```
3.6: DEFINE MOLECULE_TYPE
            Employee_TS(%DATE): ALL
     FROM   Employee_tuple
            REC_PATH
            Employee_tuple.past—Employee_tuple
            UNTIL Employee_tuple(PREVIOUS).valid
                  <= %DATE
     WHERE Employee_tuple(FIRST).future = EMPTY;
```

MAD queries deliver a set of molecules, i.e. a structured representation of the atoms involved. For the case of our TS modeling, the MAD queries deliver chains of Employee_tuple atoms. Often, however, a structured view of the result is not required. In these cases, the built-in function **MOLAGG** may be used to collect a set of attribute values within a molecule, into a list. For example, the following query delivers a list of all dates where changes have occurred in Mary's history since January 1st, 1985 (given by the values of the **valid** attribute) by aggregating them over all recursion levels (as indicated by **ALL_REC**):

† We will precede all parameters of macro definitions by a "%" sign.

```
3.7: SELECT MOLAGG
            (Employee_tuple(ALL_REC).valid)
     FROM   Employee_TS(1985/01/01)
     WHERE  Employee_tuple(FIRST).name = 'Mary';
```

Analogously, the built-in function **VALUE** converts a set of simple† molecules into a list of values. Hence, it may be used to apply aggregating functions (which are defined only on lists) to a query's result. For example, the following query retrieves all employee tuples which contain a salary value of more than the average of the salary values of all employee tuples:

```
3.8: SELECT ALL
     FROM   Employee_tuple
     WHERE  salary > AVG ( VALUE (SELECT salary
                                  FROM Employee_tuple));
```

4. Handling Temporal Queries

So far, we have introduced a temporal query language which allows for the selection and manipulation of temporal data organized as time sequences. By the means of a small set of temporal predicates (AT, DURING, WHILE), we can select temporal information in a natural way similar to the well-known SQL constructs for non-temporal data. We have shown a direct and straight-forward mapping of the basic units of temporal data (the time sequences) to complex objects provided by the MAD model. However, selecting temporal information by running MQL queries against these complex objects is much more difficult than using the temporal query language. Therefore, in the following chapter we describe the mechanism used to transform temporal queries on TS relations to MQL queries applicable to complex objects.

Transforming Temporal Queries to MQL Queries

The transformation of temporal queries essentially relies on the capabilities of the MAD model for the molecule type definition and the handling of recursion. We will use the query below as an example. It expresses that we are interested in the salaries of employees, who were working in department D12 on June 1st, 1981. Remember, the keyword **ONLY** projects data which is valid at the time specified in the restriction clause.

```
4.1: T_SELECT salary ONLY
     T_FROM   Employee
     T_WHERE  (department = 'D12') AT 1981/06/01;
```

Furthermore, we will use the history of Mary (see figure 2.1 in section 2) as a running example. The MAD model representation of the TS in figure 2.1 is constructed by the connec-

† Molecules consisting of only one atom with only one attribute.

tion of the tuples through their reference attributes **future** and **past**. For example, the **future** reference of the youngest tuple (*tuple 8*) is empty and its **past** reference points to *tuple 7*. This tuple references *tuple 8* by its **future** attribute (counter reference) and *tuple 6* by its **past** attribute. In this way the eight tuples are constituting one *TS molecule*.

To give a reply to a temporal query like the one above, it is necessary to find the tuple which keeps the information valid at a given date. Looking at the temporal query and the TS of our example, we have to use the molecule type definition *Employee_TS(1981/06/01)* as described in statement 3.6. The corresponding molecule begins with *tuple 8* and ends with *tuple 2*. Thus, the MQL statement 4.2 is the result of the transformation of the temporal query.

```
4.2: SELECT Employee_tuple(LAST) (salary)
     FROM   Employee_TS(1981/06/01)
     WHERE  Employee_tuple(LAST).valid <= 1981/06/01
            AND
            Employee_tuple(LAST).department = 'D12';
```

A *TS molecule*, or alternatively the related *Employee_TS* molecule, qualifies if it meets the following requirements expressed in the **WHERE** clause of MQL statement 4.2:

- The object under consideration was already alive at the specified time: We have to exclude those TS_Employee molecules for which the value of **valid** of the last tuple is greater than the given date (see discussion of the molecule type definition 3.6).
- The object fulfills the qualification condition of the temporal query at the given date. Thus, the attribute **department** of the last tuple of the Employee_TS molecule contains the value 'D12'.

In our example, the evaluation of the statement delivers the **salary** value (2000) of the last tuple (*tuple* 2) of the according *Employee_TS* molecule.

The mechanism of condition transformation is obvious in our example: the **AT** construct in the **T_WHERE** clause is translated to a corresponding parameter choice for the Employee_TS definition, together with the condition on the **valid** attribute[†] yielding a molecule where the last tuple reflects the interesting data. The condition of the temporal query is applied to this tuple. The other constructs allowed for qualifications of temporal queries can be transformed in similar ways. The **DURING** constructs are expanded to conditions on the **valid** attribute's value. The condition "(department = 'D03') **SOMETIMES DURING** [1980/1/1, 1980/12/31]" is transformed to the restriction clause

[†] The condition on the valid attribute could be included into the definition of Employee_TS, thereby changing the semantics of this molecule type definition.

```
WHERE EXISTS Employee_TS(ALL_REC):
      (department = 'D03') AND (valid <= 1980/12/31),
```

whereas "(department = 'D03') **ALWAYS DURING** [1980/1/1, 1980/12/31]" becomes

```
WHERE FOR_ALL Employee_TS(ALL_REC):
      (department = 'D03') OR (valid > 1980/12/31).
```

Both restriction clauses deal only with the ending point of the interval, because the starting point is already captured by the until clause of the molecule definition used in the **FROM** clause. The transformation of **DURING WHILE** is illustrated by the following example: the **T_WHERE** clause "(department ='D12') **SOMETIMES DURING WHILE** (salary>3000)" (cf. 2.4) is transformed to a MQL query containing the *Employee_TS(0/0/0)* molecule type and the following restriction clause:

```
WHERE EXISTS Employee_TS(ALL_REC):
      (department = 'D12' AND salary > 3000).
```

The various keywords allowed in the **T_SELECT** can be transformed in a similar way: A projection clause "attribute **AT** (t2)" is transformed to a qualified projection of the following shape[†] :

```
SELECT attribute
FROM   RESULT
WHERE  valid = MAX (
                   SELECT valid
                   FROM   RESULT
                   WHERE  valid <= t2 )
```

DURING [t1,t2] is mapped analogously by adding "**AND** valid <= t1" to the **WHERE** clause. The transformation of **ONLY** depends on the construct used in the WHERE clause. Query 4.1 shows an example for the combination **ONLY** / **AT**, where the relevant data is contained in the last tuple, and hence the last tuple is projected.

Up to this point, we have only discussed the transformation of temporal retrieval statements to MQL statements. Manipulation statements, however, do not create any problem, since their structure is very regular and simple. They can be directly mapped to corresponding MQL operations.

The transformation of a temporal query to an MQL query discussed so far is straight forward, because every tuple of the *TS molecule* contains all information valid at a given time. On the other hand, it is likely that only few of the attribute values of two adjacent tuples differ. This leads to high storage redundancy and requires a huge amount of storage space, which is unacceptable even with the presence of optical disks and other devices providing large storage capacity at low costs.

[†] Here, we assume that t2 > date. Otherwise, the query becomes slightly more complex.

Reducing Amount of Storage

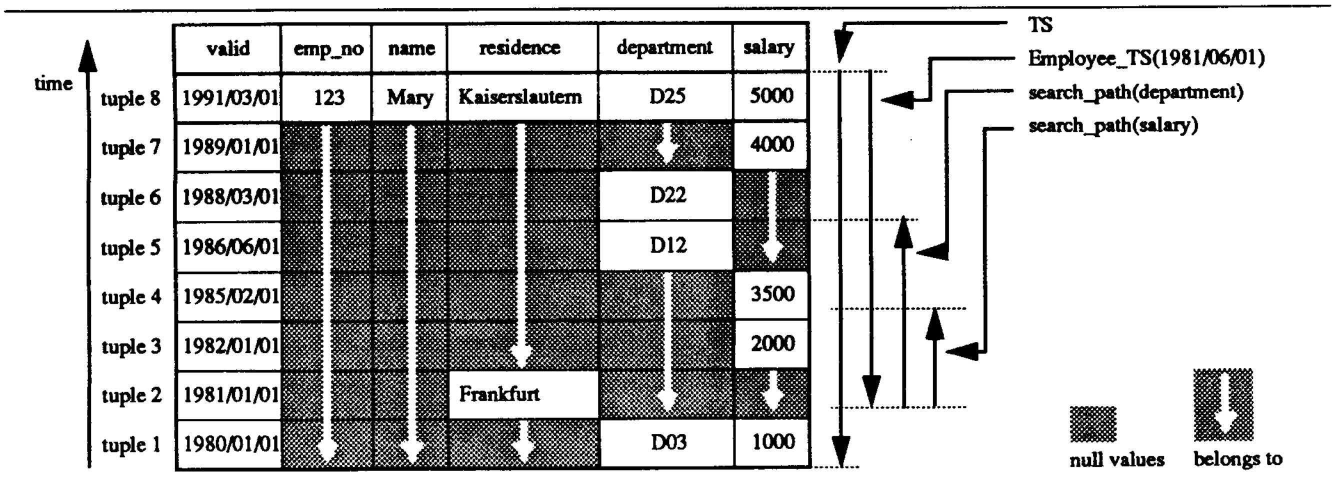
We can reduce the amount of storage space needed by storing only the differences between two adjacent tuples instead of the complete information. In [DLW84] differences on the basis of EXOR operations are used to cut storage costs. Instead of storing the full data only the result of an EXOR operation between the current tuple and the predecessor tuple (recursively continued) is stored. We refer to these differences as *reverse differences*, because the youngest tuple is completely stored whereas all older tuples are represented only by differences. The EXOR operation is only applicable at the byte level, so that the differences have to be built at the lower levels of the database management system. This has three major drawbacks. Firstly, this mechanism has to be integrated into the database management system itself. Secondly, in order to evaluate a boolean expression on one attribute, the whole tuple has to be reconstructed by traversing a chain of differences starting with the current tuple. Thirdly, the differences are "unstructured" tuples for the higher levels of the database management system. Thus, conventional access path structures cannot be used on history data. Hence, we want to evolve the method as described in [DLW84] by using logical rather than physical differences, i.e. we assign null-values to those attributes which are not altered in comparison to their temporal predecessor†. This has three advantages. Firstly, we can operate on the tuples with the data manipulation operations of the database system. Secondly, using conventional access path structures on the history data will lead us only to tuples in which the requested value is stored, but not to a huge bulk of history tuples preserving this value. Thirdly, in PRIMA, the PRototype Implementation of the MAD model [Hä88], null-values do not need any storage space.

† This implies that attributes of TS relations do not contain null values. We can extend our approach to allow for null attributes of TS relations, but this exceeds the scope of this paper.

As a consequence of using reverse differences, the tuple corresponding to the validity time specified does not necessarily contain the values of all attributes being included in the projection list or the qualification condition of the temporal query. For that reason, we have to extend our mapping algorithm in order to find the attribute values which belong to a certain validity time but are not stored in the according tuple.

Figure 4.1 shows the TS of our example on the basis of reverse differences. The empty fields represent the null values. The arrows indicate to which tuples a stored value also belongs. To evaluate the qualification condition on the last tuple of the *Employee_TS* molecule in our temporal query (*tuple 2*), we have to find the value of the attribute **department**, which was current while *tuple 2* was valid. We find the information in *tuple 5*. The concept of reverse differences forces the complete storage of the information in the youngest time tuple with **alive** = ***TRUE***†. This guarantees that a defined value is found when traversing the chain of tuples using the **future** references. In order to facilitate the search for the attribute value, we define a new molecule type *search_path* (MQL statement 4.3), which starts with a time tuple and examines all tuples along the **future** references until the first one with a defined value in the specified attribute is found. The name of this attribute is the parameter of the molecule type definition. To detect the first defined value we use the MAD operator **IS_NULL**. It delivers **TRUE** if the given attribute has a null value, **FALSE** otherwise. Thus, we can find the relevant value of the attribute **department** in the last tuple of the molecule *search_path(department)*, if we choose a proper starting point. In the same way we define the molecule *search_path(salary)* to look for the value of the projection attribute. Examples of *search_path* molecules are illustrated in figure 4.1.

† A "gravestone" contains only one defined attribute value: alive = FALSE



	valid	emp_no	name	residence	department	salary
tuple 8	1991/03/01	123	Mary	Kaiserslautern	D25	5000
tuple 7	1989/01/01					4000
tuple 6	1988/03/01				D22	
tuple 5	1986/06/01				D12	
tuple 4	1985/02/01					3500
tuple 3	1982/01/01					2000
tuple 2	1981/01/01			Frankfurt		
tuple 1	1980/01/01				D03	1000

Figure 4.1: TS, Employee_TS and search_path molecules in the case of reverse differences

```
4.3: DEFINE  MOLECULE_TYPE
             search_path(%ATTRIBUTE) : ALL
     FROM    Employee_tuple REC_PATH
             Employee_tuple.future—Employee_tuple
             UNTIL NOT IS_NULL
          (Employee_tuple(PREVIOUS).%ATTRIBUTE);
```

We find the proper starting points of the *search_path* molecules by connecting them to the *Employee_TS* molecules using a molecule join. The first part of the restriction clause of MQL statement 4.4 represents the join condition, ensuring that the last tuple of the *Employee_TS* molecule and the first tuples of the *search_path* molecules have to be identical. To be able to address both *search_path* molecule types in the projection clause or the restriction clause of the MQL statement, we assign the role names *D* and *S* to them.

```
4.4: SELECT  MOLAGG
             (S(Employee_tuple(LAST)(salary)))
     FROM    Employee_TS(1981/06/01),
             D(search_path(department)),
             S(search_path(salary))
     WHERE   (Employee_TS.Employee_tuple(LAST).ID
              =D.Employee_tuple(FIRST).ID)
     AND  (Employee_TS.Employee_tuple(LAST).ID
          = S.Employee_tuple(FIRST).ID)
     AND  (Employee_TS.Employee_tuple(LAST).valid
          <= 1981/06/01)
     AND  (D.Employee_tuple(LAST).department
          = 'D12') ;
```

To qualify, the joined molecule has to meet the following requirements:

- The corresponding object must have been alive at 1981/06/01. This is true, if **valid** of the last tuple is equal or less than this date.
- The qualification condition on **department** must be evaluated to be **TRUE** with respect to the value found in the last tuple of the *search_path(department)* molecule.

If these conditions are fulfilled, the value of the attribute **salary** in the last tuple of the *search_path(salary)* molecule can be delivered, as $2000 in the example.

MQL statement 4.4 is rather complex and poses high requirements on the query optimizer in order to generate an efficient query plan. Fortunately, PRIMA has been designed with respect to extensibility, i.e. it is possible and quite easy to integrate new operators into the query language [SS90]. Here, we use this facility to define a new operator in order to facilitate the formulation and the computation of these types of queries.

Using the Advantages of Extensible Database Systems

MQL statement 4.4 shows that the use of reverse differences could force very complex computations. For every attribute in the projection list and the qualification condition, a *search_path* molecule has to be created in order to find the according value. The correct combination and the efficient evaluation of complex conditions containing several attributes will pose strong demands on a query optimizer. The specification of the attribute computation by specific molecule types for each attribute leads to a waste of join conditions. The desired operation, however, is not a join, but a simple back-traversing of the molecule which is already known. Furthermore, this back-traversal is not necessary for each attribute separately, but could be done for all attributes in one traversal. The complex query formulation is only enforced by the lack of a reverse difference operator in the query language. Hence, we employ the extensibility of PRIMA to define and integrate a new operator into the query language [SS90].

For the purpose of reverse difference computation the new operator, called **REV_DIFF** should work in a similar way to the **MOLAGG** operator. However, instead of collecting all attribute values into a list, **REV_DIFF** is designed to deliver a list of the values of the specified attribute in the deepest recursion level where they are defined. It will always be called in combination with the keyword **ALL_REC** to indicate that it works on the whole recursive molecule. This is obviously only a slight modification of the **MOLAGG** operator. In our application, **REV_DIFF** will always deliver a list with exactly one element. MQL statement 4.5 is an improved version of MQL statement 4.4 using the **REV_DIFF** operator.

```
4.5: SELECT  MOLAGG (REV_DIFF
             (Employee_tuple(ALL_REC).salary))
     FROM    Employee_TS(1981/06/01)
     WHERE   (Employee_tuple(LAST).valid
             <= 1981/06/01)
       AND  (REV_DIFF
       (Employee_tuple(ALL_REC).department) = 'D12');
```

Now, we are able to omit the complex molecule joins and the *search_path* molecules. Thus, a lot of difficult problems arising for query optimization from statement 4.4 vanish when executing statement 4.5. Further advantages are the simpler handling of information stored with reverse differences and the easier transformation of temporal queries to MQL statements.

So far, we have shown the transformation of temporal queries to MQL queries on complex objects, even in the case of using reverse differences. Besides the saving of storage amount, we want to optimize the access time to the present data.

Reducing Access Time

There is a common belief in the literature [AS86b, AS88, DLW84, SK86], that there is a correlation of the access frequency and the age of data. Furthermore, it is postulated that the performance of temporal database management systems processing the present data must be almost as good as that of conventional database management systems. In order to achieve this goal we have to separate the current data from the bulk of history data [Lu84, RS87]. This leads to the modeling of a temporal relation by two different atom types (see Appendix). In this approach each TS *molecule* is composed of one occurrence of an *_anchor* atom type (in general representing present data) and an arbitrary number of occurrences of a *_tuple* atom type. To connect the *_anchor* atom with the first *_tuple* atom we additionally need the reference attribute **hist** (*_anchor* atom type) and the counter reference attribute **present** (*_tuple* atom type).

As a consequence of modeling temporal data by the use of two different atom types we expand a temporal query to two different MQL statements. The first one processes the actual data and touches only the *_anchor* atom type (MQL statement 4.6). The second one processes only those *TS molecules* which keep the relevant information in the history tuples and have not been investigated during the evaluation of the first statement (Appendix, MQL statement A.2).

```
4.7: SELECT  Employee_anchor(salary)
     FROM    Employee_anchor
     WHERE   (Employee_anchor.valid <= 1981/06/01)
             AND (Employee_anchor.department = 'D12')
             AND (Employee_anchor.alive = TRUE);
```

Statement 4.6 is a very simple MQL statement. It works on the atom type *Employee_anchor* and does not care about complex *TS molecules*. To check the temporal relevance of an *Employee_anchor* atom we have to check two conditions. Firstly, the given date belongs to the validity duration of the anchor atom (*valid <= 1981/06/01*). Secondly, the anchor tuple is not a gravestone (*alive = TRUE*), i.e. the object is still alive.

5. Conclusions and Outlook

In the first part of the paper, we have introduced a temporal data model which serves as a framework for the following considerations. It can be viewed as an extension of the relational model in the temporal dimension. Relations are extended to time sequence relations by extending each tuple to an ordered list of tuples, called time sequence. A change to an attribute value is reflected by the insertion of a new tuple into the time sequence. Thus, a time sequence represents the evolution of an entity during its lifetime. We have defined several operations to manipulate time sequences. For the retrieval in our temporal data model, we offer juncture queries, range queries and coincidence queries.

In the recent years, many efforts have been made to model temporal data by means of the relational model. Nevertheless, all these approaches suffer from the incapacity of the relational model to cope with complex structured information, e.g. ordered lists of tuples. This is the main reason why temporal coincidences cannot be directly expressed by corresponding relational queries. In contrast to that, complex object data models offer more powerful facilities for this purpose. The MAD model chosen in our approach allows for modeling a time sequence by a complex object, thereby preserving the coherency of the tuples of this time sequence. We have detailed that the operations of our temporal data model can be easily transformed to MAD model operations, taking advantage from the MAD model's facility to handle a complex object as a unit, even in the case of a recursively structured object.

The MAD model's properties can be used to solve a further problem imposed by the management of temporal data: Since all updated data are explicitly preserved, a huge amount of storage space is required. The mechanism of "reverse differences" has been developed to store only the changing data. We can easily integrate a variant of this mechanism into our MAD model approach. The introduced MAD operations can be extended to handle reverse differences within the corresponding database management system PRIMA. Since the null values used by the difference mechanism do not require any storage space in PRIMA, the volume of data to be stored can be reduced considerably. Access time to frequently used parts of the temporal data can be optimized by separating them from the rest of the temporal data. We have discussed an extension of our approach to integrate this improvement, too. It furthermore allows us to define access paths and clustering separately for both parts of the data.

Due to the mapping of the temporal data model to a complex object data model, the processing of temporal data can be supported by the access paths offered by the complex object data model. In the case of our approach, a B*-tree defined on the data stored with the reverse difference mechanism would lead us to all tuples where the values of the corresponding attribute had changed. Our future work includes a detailed investigation of this issue: When are conventional access paths structures helpful in our approach? Which kind of queries can be supported by them? Are dedicated complex object access paths useful in our framework?

We have presented a method to establish a temporal "relational" database system on top of a complex object database system. The principles depicted in our paper cannot only be applied to the MAD model, but also to other complex object data models. Some of the extensions like the partitioning of data according to the access characteristics, and the usage of reverse differences, however, are not supported by all other complex object data models and the corresponding database management systems.

Finally, the question arises whether a temporal extension of a complex object data model can be modeled using a complex object data model again. Particularly in the case where the complex objects may share components, there is a new quality of the problem. In contrast to the temporal extension of the relational model, we need not only cope with the evolution of relations (i.e., of their attribute values), but also with the evolution of relationships. This introduces a lot of interesting new problems, which we will investigate in the future.

6. References

Ar86: Ariav, G.: A Temporally Oriented Data Model, ACM TODS, Vol. 11, No. 4, 1986, pp. 499-527.

AS85: Ahn, I., Snodgrass, R.: A Taxonomy of Time in Databases, Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, 1985, pp. 236-246.

AS86a: Ahn, I., Snodgrass, R.: Temporal Databases, IEEE COMPUTER, Vol. 19, No. 9, 1986, pp. 35-42.

AS86b: Ahn, I., Snodgrass, R.: Performance Evaluation of a Temporal Database Management System, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington, 1986, pp. 96-107.

AS88: Ahn, I., Snodgrass, R.: Partitioned Storage for Temporal Databases, Information Systems, Vol. 13, No. 4, 1988, pp. 369-391.

Bo82: Bolour, A., Anderson, T.-L., Dekeyser, L.-J., Wong, H.-K.-T.: The Role of Time in Information Processing: A Survey, ACM SIGMOD RECORD, Vol. 12, No. 3, 1982, pp. 27-50.

CW83: Clifford, J., Warren, D.-S.: Formal Semantics for Time in Databases, ACM TODS, Vol. 8, No. 2, 1983, pp. 214-254.

Da86: Dadam, P., et al.: A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat tables and Hierarchies, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington, 1986, pp. 356-367.

DLW84: Dadam, P., Lum, V., Werner, H.-D.: Integration of Time Versions into a Relational Database System, Proc. 10th Int. Conf. on VLDB, Singapore, 1984, pp. 509-522.

Ga88: Gadia, S.-K.: A Homogeneous Relational Model and Query Language for Temporal Databases, ACM TODS, Vol. 13, No. 4, Dec. 1988, pp. 418-448.

Hä88 Härder, T.: Overview of the PRIMA Project, in: Härder, T. (ed.): The PRIMA Project - Design and Implementation of a Non-Standard Database System, Research Report No. 26/88, University Kaiserslautern, 1988, pp. 1-12.

Kä90: Käfer, W.: The Temporal Query Language TMQL, Internal Report, University Kaiserslautern, 1990.

Kl81: Klopprogge, M.-R.: TERM An Approach to Include the Time Dimension in the Entity-Relationship Model, Proc. 2nd Int. Conf. on Entity-Relationship Approach, 1981, pp. 477-512.

Lu84: Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., Woodfill, J.: Designing DBMS Support for the Temporal Dimension, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, 1984, pp. 115-130.

Mi88 Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, Proc. 2nd Int. Conf. on Expert Database Systems, Tysons Corner, 1988, pp. 33-50.

Mi89 Mitschang, B.: Extending the Relational Algebra to Capture Complex Objects, Proc. 15th Int. Conf. on VLDB, Amsterdam, 1989, pp. 297-305.

RS87: Rotem, D., Segev, A.: Physical Organization of Temporal Data, Proc. Int. Conf. on Data Engineering, IEEE Computer Society, Los Angeles, 1987, pp. 547-553.

Schö89: Schöning, H.: Integrating Complex Objects and Recursion, Proc. of the 1st Int. Conf. on Deductive and Object-Oriented Database Systems, Kyoto, Japan, 1989, pp. 535-554.

SK86: Shoshani, A., Kawagoe, K.: Temporal Data Management, Proc. 12th Int. Conf. on VLDB, Kyoto, Japan, 1986, pp. 79-88.

Sn86: Snodgrass, R.: Research Concerning Time in Databases: Project Summaries, ACM SIGMOD RECORD, Vol. 15, No. 4, 1986, pp. 19-39.

Sn87: Snodgrass, R.: The Temporal Query Language TQuel, ACM TODS, Vol. 12, No. 2, 1987, pp. 247-298.

SS86: Schek, H.-J., Scholl, M.-H.: The Relational Model with Relation Valued Attributes, Information Systems, Vol. 11, No. 4, 1986, pp. 137-148.

SS90: Schöning, H., Sikeler, A.: Extending and Configuring the PRIMA Database Management System Kernel (extended abstract), in: Proc. PARBASE-90, Miami Beach, Florida, 1990.

Ta86: Tansel, A.-U.: Adding Time Dimension to Relational Model and Extending Relational Algebra, Information Systems, Vol. 11, No. 4, 1986, pp. 343-355.

7. Appendix

In order to optimize the access time to the current dates data, we have to split the TS molecules into two parts (cf. section 4). The first part contains the newest tuple of a TS, i.e. the tuple with **future** = **EMPTY**. In general, this will be the current's date data[†], and will be stored in an atom type with the suffix *_anchor*. The rest of the data is stored in the atom type *_tuple* as described above. The *_anchor* atom type will contain only a small part of the overall data. This splitting of the data requires two additional reference attributes in the corresponding MAD schema in order to link the two atom types. The *_tuple* atom with an empty **future** reference attribute will be linked to the corresponding *_anchor* atom by reference attribute **present**.

The definition of the TS molecule (A.1) based on such a schema is analogous to statement 3.6. Since we know that the anchor atom is the latest tuple, we can omit the condition "future = EMPTY" here, saving an exhaustive search for tuples with an empty reference attribute in order to find the proper molecule root.

```
A.1: DEFINE MOLECULE_TYPE Employee_TS(date): ALL
     FROM Employee_anchor.hist—Employee_tuple
              REC_PATH Employee_tuple.past—Employee_tuple
              UNTIL     Employee_tuple(PREVIOUS).valid
                        <= %date;
```

Statement A.2 is the complementary statement to statement 4.6. Here, we have to consider only those TS molecules for which the *_anchor* atoms do not contain the data we are looking for. The shaded part of statement A.2 marks the corresponding part of the WHERE clause. In the case of a search for current date's data we will only touch a small percentage of all TS molecules...

```
A.2: SELECT MOLAGG
            (REV_DIFF (Employee_tuple(ALL_REC).salary))
     FROM   Employee_TS(1981/06/01)
     WHERE  (Employee_tuple(LAST).valid <= 1981/06/01)
            AND NOT (Employee_anchor.valid <= 1981/06/01)
            AND (REV_DIFF
            (Employee_tuple(ALL_REC).department = 'D12') ;
```

[†] The _anchor atom type may also contain tentative data (belonging to the future), in which case the current date's data is stored in the history chain. Gravestones are also stored in the _anchor atom type.