in: Workshop Tolouse, 1991

Temporal Queries to Complex Objects

Wolfgang Käfer, Harald Schöning
University Kaiserslautern
Germany
email: {kaefer, schoenin}@informatik.uni-kl.de

Abstract

Support for temporal data continues to be a requirement posed by many applications such as VLSI design and CAD, but also in conventional applications like banking and sales. The strong demand for complex-object support is known as an inherent fact in design applications, but it also holds for advanced "conventional" applications. Thus, new advanced database management systems should include both features, i.e. should support temporal complex-objects. We show that such a system can be efficiently implemented on top of a (non-temporal) complex-object data model. The central notion of the temporal complex-object data model is a *time slice*, representing one state of a complex object. Time slices cannot be directly stored, if non-disjunct (i.e. overlapping) complex objects are allowed. We explain the mapping of time slices onto the complex objects supported by the MAD model. Operations on temporal complex-objects are easily transformed into MAD model operations. Furthermore, we reduce the huge storage requirements usually arising from temporal databases.

1. Introduction

All human activities are embedded in time. Hence, the model of the world which is used to describe relevant facts in a database also should be capable of including temporal aspects. However, commercial databases lack this feature, and instead only show the latest state of the world. Concerning a temporal extension to the relational data model, there is plenty of literature, e.g. [SK86, Sn86, Ta86, Ga88]. Generally, these proposals do not represent the history of an entity as a whole, but rather cut it into several pieces (separate tuples). The implementation of a temporal data support for the relational data model presented in [KRS90] overcomes this restriction by treating the history of an entity (represented by a relational tuple) as a complex object. This allows for powerful yet simple retrieval operations and for an efficient implementation of the system by mapping it onto a complex-object database system.

Having found a satisfying solution for the temporal extension of the relational data model, we now raise the question of whether there is a similar solution which allows for a temporal extension even of the complex-object data model itself, which in [KRS90] was used only to realize the relational extension. The request for complex-object supporting database systems originates from various design areas. In order to employ database systems for their applications, designers need powerful modelling facilities. One of the most important requirements is that of modelling a design object as one (complex) object in the database, which can be accessed as a whole. However, complex-object data models are useful not only in these areas, but also in almost all areas, where the relational model has been successful. This fact will be illustrated by our running example which models a small part of the business world, consisting of some companies with their employees. Furthermore, design environments (and also almost all other ap-

plication areas) are characterized by a continual development of their data. This poses at least two requirements to database systems supporting these areas successfully:

- The database system has to supply the notion of a *version*, which is a state of design which the designer wants to fix explicitly, in order to be able to refer to it later on or to pass it to a cooperating designer. These aspects of temporal data are covered in [KS91] and [Kä91].
- The database system has to supply the notion of the *history* of a complex object, e.g. in order to enable the reviewing of a design process or to keep track of the development of a company with its projects and employees etc. This is orthogonal to the previously mentioned versioning concept, because here nobody knows in advance, which facts in the history of the complex objects will be of any interest in the future. Furthermore, the semantics of a version (i.e. a distinguished (intermediate) result or unit of cooperation) do not apply to the history aspect.

In this paper, we deal with history management for complex objects. We choose the molecule-atom data model (MAD model) [Mi88], which is a general complex-object data model allowing for overlapping complex objects, to serve both as the data model to be extended by a temporal dimension and as the basis for the implementation of this extension.

2. Motivation

We will implement the temporal extended MAD model (for short the TMAD model) by mapping it onto the MAD model, i.e. by performing transformations as depicted in Figure 1. The MAD model itself is implemented in the PRIMA system [HMMS87]. The approach of transforming temporal data model objects and operations has already shown to be an efficient means for the implementation of a temporal extended relational model, which is described in [KRS90]. However, we face new difficulties here which are caused by the higher complexity of the objects to be dealt with and the dynamism in the object type definition provided by the MAD model.

- In contrast to tuples of the relational model, the (complex) objects we deal with in TMAD are defined at query time (and not in the database schema).
- Whenever a tuple changes, we can store the new tuple and preserve the old one (perhaps, applying some difference techniques in order to save storage space). The change to a complex object, however, cannot be determined, because complex objects may overlap due to the dynamism of their definition. The only thing we can record is the change to tuples (and the links among them).

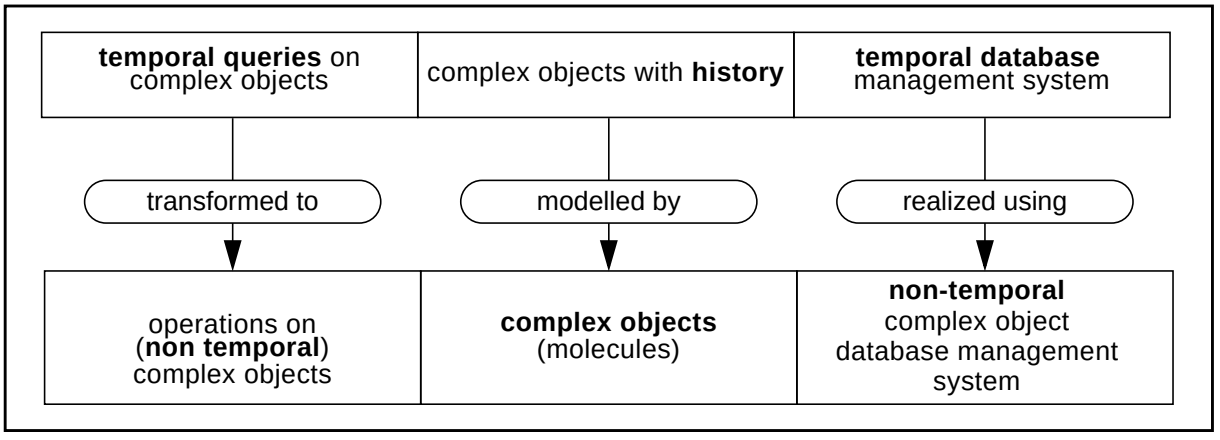

Figure 1: The mapping of the temporal data model to the complex-object data model

Our approach may be characterized as follows. Whenever a complex object change is performed, this is recorded as a change to tuples and links. The retrieval procedure is much more complicated. It bases on the notion of time slices. A time slice is defined as a time interval which contains only one state of each tuple of the complex object. For each time slice, the corresponding complex object is built up. Afterwards, the specified conditions and time restrictions are checked in order to identify the qualifying objects. If such objects exist, a projection is evaluated on the whole history of the complex object, i.e. all its time slices. We introduce an additional layer on top of the MAD model (called temporal layer) which is responsible for the transformation of the database schema and queries of the temporal data model into those of the MAD model. Furthermore, this layer has to construct time slices from the molecules delivered by the MAD model.

Section three gives a brief description of the MAD model and introduces our running example. The query facilities of the model are exemplified on a small sample database.Section four deals with the temporal extensions of the MAD model thereby defining the TMAD model. The implementation of the TMAD model by means of the MAD model is shown in section five. We close with a short summary and some conclusions.

3. The MAD model

Before we start with the introduction of the main concepts of the MAD model, we depict a small part of a business world, which will serve us as a running example throughout the paper. *Companies* organize their work in *Projects*, which may be shared with other companies. *Employees* are assigned to at least one project. Each project has a single manager. Furthermore, there are *sport clubs* which are sponsored by the companies and therefore are open only to their employees. In this small world, we can identify several complex-object types: A project corresponds to a complex-object type consisting of project-specific information and all employees assigned to that project. Obviously, instances of this type overlap, if an employee is assigned to more than one project. A company also forms a complex-object type, including projects and employees.

Another example of a complex-object type is the club (consisting of members and club-specific data). Even *employee* can be seen as a complex-object type, consisting of personal data, project assignment, and club membership. From this enumeration, one can conclude that complex-object types belonging to the same mini-world are non-disjoint and that it is not predictable which of the possible complex-object types serve the specific needs of an application. A complex-object data model reflecting these observations is the Molecule-Atom Data Model (MAD Model) [Mi88], which allows for dynamic complex-object type definitions at query time.

Thus, the database schema consists of a network of building blocks (called atom types) which directly reflects an entity/relationship model of the mini-world†. Such a schema which describes the mini-world introduced above is shown in Figure 2. Relationships are modelled by pairs of attributes of type REF_TO. For example, the relationship between *Club* and *Employee* is modelled by the REF_TO attributes *Club.members* and *Employee.club*. REF_TO attribute definitions may be augmented by cardinality restrictions which indicate the minimal and the maximal number of identifiers contained in the attribute's value. A * indicates "no restriction on the maximal number of identifiers". For example, a club must have at least four members, and an employee works exactly for one company according to the

† with binary attribute-free relationships

schema depicted in Figure 2. Each instance of an atom type (called atom) has a unique system-controlled identifier. A value of a REF_TO attribute consists of a set of such identifiers. To indicate that atoms *a* and *b* are related (linked), the REF_TO attribute of *a* which points to *b*'s type contains *b*'s identifier, and additionally, the value of the corresponding REF_TO attribute of *b* contains *a*'s identifier.

```
CREATE ATOM_TYPE  Company (co_id:   IDENTIFIER,
                  company_name:     STRING,
                  employees:        REF_TO (Employee.company) (1,*),
                  projects:         REF_TO (Project.companies) (1,*));
CREATE ATOM_TYPE  Employee (emp_id: IDENTIFIER,
                  employee_name:    STRING,
                  salary:           INTEGER,
                  company:          REF_TO (Company.employees) (1,1),
                  works_for:        REF_TO (Project.employees) (1,*),
                  manager_of:       REF_TO (Project.managed_by) (0,1),
                  club:             REF_TO (Club.members) (0,*));
CREATE ATOM_TYPE  Project (proj_id: IDENTIFIER,
                  project_name:     STRING,
                  employees:        REF_TO (Employee.works_for)(1,*),
                  managed_by:       REF_TO (Employee.manager_of) (1,1),
                  companies:        REF_TO (Company.projects) (1,*));
CREATE ATOM_TYPE  Club (club_id:    IDENTIFIER,
                  club_name:        STRING,
                  club_desc:        STRING,
                  members:          REF_TO (Employee.club)(4,*));
```

Figure 2: Database schema of the sample database

Using the MAD model's query language MQL, one can retrieve the complex-object company "Cheese" (including projects and employees) by the following query:

```
SELECT    ALL
FROM      Company.projects-Project.employees-Employee
WHERE     company_name = "Cheese"
```

The query is evaluated by searching qualifying *Company* atoms, adding all *Project* atoms whose identifiers are contained in the *projects* REF_TO attribute, and adding the *Employee* atoms whose identifiers are elements of their *employees* REF_TO attribute.

The SELECT clause specifies which parts of the complex object shall be shown (projection clause), the FROM clause describes the complex-object type to work on, and the WHERE clause is used to specify which complex objects of that type qualify.

The query only includes information about the employees, but not about the managers of the projects. We can change the query to obtain this information by following two paths to *Employee*. To be able to distinguish the two occurrences of *Employee* in this query, we assign so-called role names (E and M) to them:

```
SELECT    ALL
FROM      Company.projects-Project-(  .employees-E(Employee),
                                      .managed_by-M(Employee))
WHERE     company_name = "Cheese"
```

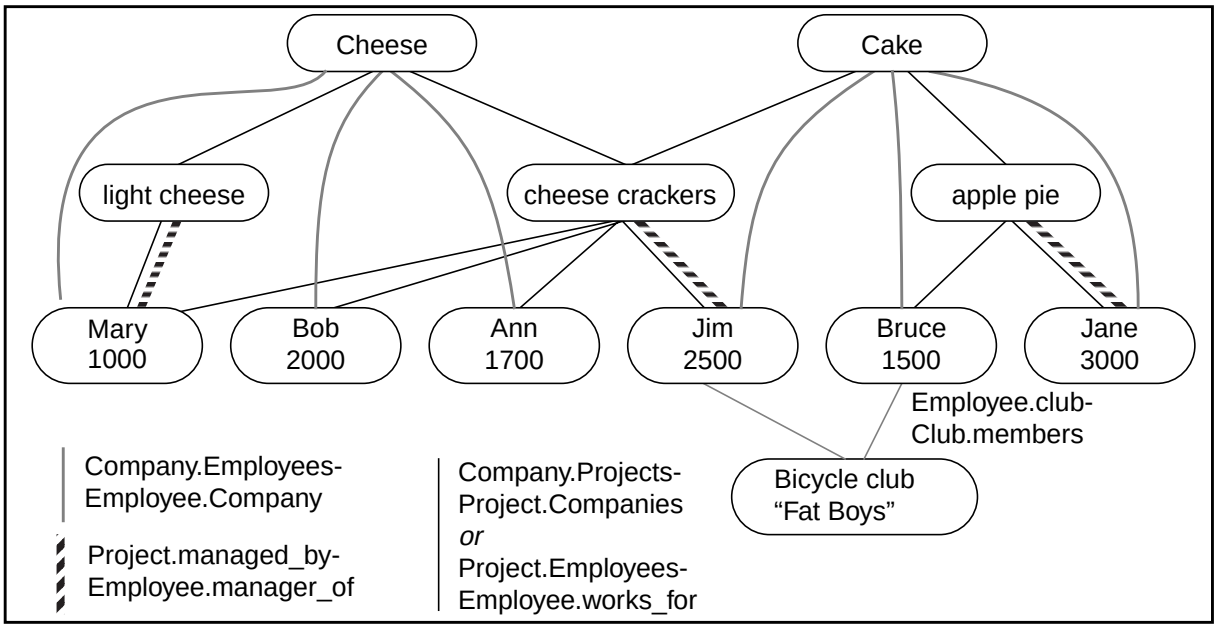

Figure 3: Sample database (valid at the first of June, 1988)

A sample database is shown in Figure 3. The “bubbles” represent atoms, whereas the lines represent the links between them. Regarding this database, we notice that the query shown above will include the projects “cheese crackers” and “light cheese” and the employees “Mary”, “Bob”, “Ann” and “Jim” into the complex object “Cheese”. If we want to include only those projects, which exclusively belong to the company, we have to rewrite the projection clause (to perform a so-called “qualified projection”):

```
SELECT   Company (ALL), SELECT   Project(ALL), Employee(ALL)
                        FROM     RESULT
                        WHERE    COUNT(Project.companies) = 1
FROM     Company-Project.employees-Employee
WHERE    company_name = “Cheese”†
```

The keyword RESULT indicates that the corresponding SELECT query refers to the result obtained by the surrounding query. ALL leads to the projection of all attributes of the corresponding atom type. Thus, this query selects only those projects which belong to exactly one company (i.e. to the company we had started with).

Furthermore, the MAD model can handle recursively structured complex objects. The expanded database schema as shown in Figure 7 contains two additional attributes of the REF_TO type for each atom type of the original schema: *past* and *future* are used to establish a chain of atoms of the same atom type. We can retrieve these recursively structured molecules by the following query:

```
SELECT   ALL
FROM     Employee REC_PATH Employee.past-Employee
         UNTIL salary< 3000
WHERE    Employee(FIRST).employee_name= “Mary” AND
         Employee(FIRST).future = EMPTY;
```

† The name of a REF_TO attribute may be omitted, if there is only one such attribute connecting the two atom types.

The keyword REC_PATH forces the construction of recursive complex objects, which is stopped when the condition contained in the UNTIL clause would be fulfilled. The algorithm starts with an *Employee* atom fulfilling the conditions indicated in the WHERE clause†. *Employee(FIRST)* names the first *Employee* atom in the molecule (i.e. the root of the molecule). Then, the *Employee* atom 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, in our example, the recursion terminates whenever a salary less than 3000 is encountered or when the *past* attribute is empty.

2/1/1989:	Mary earns 2000
3/1/1989:	Mary quits from project “cheese crackers”
4/1/1989:	Ann quits company “Cheese” and joins “Cake”
	John joins company “Cheese” and is assigned to Project “light cheese”
5/1/1989:	Bob becomes Manager of “cheese crackers” instead of Jim
6/1/1989:	Bob becomes member of bicycle club “Fat Boys”
7/1/1989:	A new project “lemon tart” is created by company “Cake”
	Bruce changes its project to “lemon tart” and becomes its manager
	Ann is assigned to “lemon tart” and “cheese crackers”

Figure 4: Changes applied to the sample database during the year 1989

We cannot detail the MAD model’s powerful operations here. The interested reader is referred to [Mi88, Schö89].

Obviously, the complex objects as depicted in Figure 3 are subject to changes: new employees are hired, salaries are raised, project assignments change, etc. We have recorded some changes in Figure 4. Unfortunately, the MAD model discussed so far is capable only of showing the latest state of the database. Figure 5 shows the database state after all these changes have been performed.

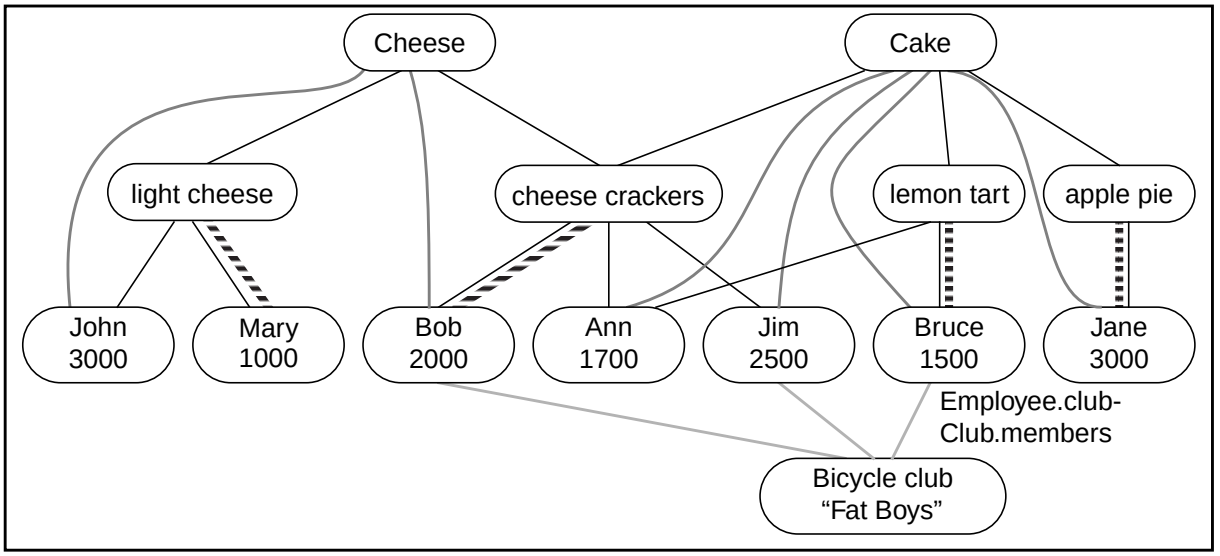

Figure 5: Sample database (valid at December 31 of 1989)

† *Employee(FIRST).future = EMPTY* ensures, that the recursion process starts with the most recent *Employee* atom.

‡ In our application, the atom chain established by the *past/future* references is cycle-free. Otherwise, the recursion would terminate whenever a cycle would appear in the molecule.

There is no support for a temporal dimension which would allow for querying the history of a complex object. Many applications, however, require just this feature. Typically, queries like the following occur:

- "How did company *Cheese* look like at 3/2/1989?"
- "Which clubs had members of company *Cheese* in the year 1989?"
- "Who has ever been employee of company *Cheese*?"
- "For which projects did Mary work during the whole year 1989?"

In the following, we present a temporal extension to the MAD model and MQL, which supports these kinds of queries. Then, we will discuss how this extension can be implemented on top of the MAD model.

4. The Temporal Model

As mentioned above, the MAD model's UPDATE operation works like in other non-temporal database management systems by overwriting the previous values with the new data. Thus, the database represents only the actual data of the related mini-world like a snapshot. All earlier snapshots of the database are lost and cannot be retrieved. In contrast to that, temporal databases preserve all snapshots of the database and allow a wide variety of retrieval facilities on the actual and historical data. In the following, we will describe the extensions to the schema definition which are necessary to capture all snapshots of the database, and the operations which are necessary to handle the historical data. We record each state of an atom by storing the attribute values of each atom's state together with the time interval of the state's validity. For this purpose, we add the attributes *valid_from* and *valid_until* to each atom type definition. Furthermore, all states of an atom are connected to one another by the attributes *past* and *future* (cf. Figure 7). In the sequel, we will use the term "temporal atom" for the collection of all states of an atom.

```
CREATE ATOM_TYPE Company(co_id:     IDENTIFIER,
                         company_name:  STRING,
                         employees:     REF_TO (Employee.Company) (1,*),
                         projects:      REF_TO (Project.Companies) (1,*),
                         valid_from:    TIME(DAY),
                         valid_until:   TIME(DAY),
                         past:          REF_TO (Company.future) (0,1),
                         future:        REF_TO (Company.past) (0,1));
CREATE ATOM_TYPE Employee(...
CREATE ATOM_TYPE Project(...
CREATE ATOM_TYPE Club (...
```

Figure 6: Schema extension of the database in order to capture the historical data

Temporal Databases and Updates

In order to preserve the previous values of the data which is usually overwritten by an UPDATE operation, we have to redefine the semantics of this operation. The new operation T_UPDATE† has to work in a different way. Instead of overwriting the previous values, T_UPDATE creates a copy of the original atom, changes the related values and inserts the modified atom into the database. Now we have to con-

† We precede all operations of our temporal model with T_.

sider two points, because these two atoms logically represent one atom. Firstly, we have to mark each atom with its validity interval. Secondly, we have to combine the atoms which build one temporal atom.

We accomplish the first task by assigning appropriate values to the attribute pair *valid_from* and *valid_until*† of the atom. The T_UPDATE operation has to ensure that the *valid_from* attribute of the new atom contains the same value as the *valid_until* attribute of the previous atom in order to avoid "holes" in the history of the atom. The user has to specify the validity time in the T_UPDATE statement, because this time reflects when the associated fact will be valid in the mini-world. There is also the notion of transaction time, which represents the time, when a fact is recorded in the database. In the following, we consider only the validity time, because the transaction time can be handled analogously.

In order to accomplish the second task mentioned above, we have to combine all atoms constituting the history of one (temporal) atom. We do this by creating a time ordered chain of these atoms via REF_TO attributes (named *past* and *future*). Thus, each atom representing a part of a history contains in its *past* attribute the identifier of the previously valid atom and in its *future* attribute the identifier of the next newer atom, respectively. In the following, we call such a chain of atoms *Time Sequence* (TS) [KRS90].

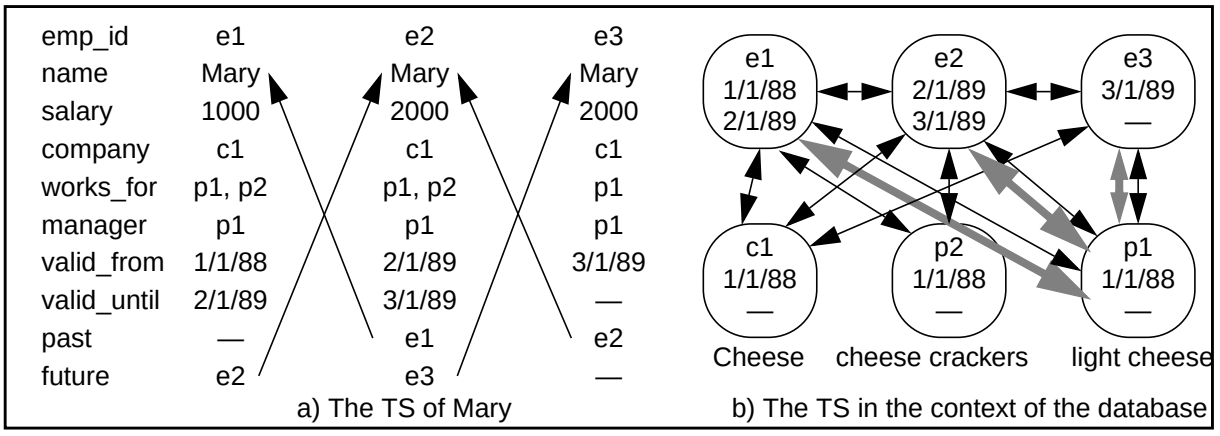

Figure 7: History of Mary

As an example, Figure 7a shows the TS representing Mary's history according to our mini-world. The TS of Mary consists of three atoms. The most recent atom contains no reference in its *future* attribute, whereas the oldest atom contains no reference in its *past* attribute. Furthermore, Figure 7b reflects the fact that Mary's history is only one part of the database and therefore the atoms are connected with other atoms of the database constituting the so-called complex objects (the whole database is shown in Figure 8). Raising Mary's salary to 2000 at the 1st of February 1989 forces the insertion of a new atom into the database as described by the T_UPDATE operation. Thus, all references (i.e. all values of the REF_TO-type attributes) are copied into the new atom connecting it to all previously referenced atoms. In order to guarantee referential integrity, the database system modifies the referenced atoms making them reference also the new atom of Mary's history (i.e. of Mary's TS) [Schö90]. Please notice, that this is the only real update operation in our scenario‡. Of course, we could reflect the modifications to the REF_TO attributes by creating a new atom as we do for other updates. Then, references to the new atom would

† *valid_until* contains a special NULL value, if we don't know how long the data will be valid in the future.

‡ Obviously, our approach leads to different databases representing the same mini-world. If, for example, Bob joins the bicycle club "Fat Boys", this can be seen as a change to Bob's state (creating a new atom representing Bob and updating the references of the "Fat Boys" atom). Or it can alternatively be seen as a change to "Fat Boys", creating a new atom of the bicycle club and updating Bob's references. The retrieval procedure discussed in section 5, however, constructs the same result from both databases for each query.

have to be added to all atoms referencing it. Thus, a snowball effect would be initiated, which in the worst case could generate a new copy of almost the whole database.

As a consequence of our approach, cardinality restrictions have to be reconsidered. The upper bounds may not longer hold, if for an atom references to more than one atom of the same TS exists. For example, in the database shown in Figure 8, the REF_TO attribute *managed_by* of "cheese crackers" contains 3 identifiers, although in the original schema the cardinality restriction was (1,1). Hence, we remove the upper bounds when transforming the original schema.

Summarizing, we should stress that the extension of each atom type definition of the database schema by two attribute pairs (*valid_from*/*valid_until* and p*ast/future*) and the modification of the update operation is sufficient for storing the historical data along with the actual data in the database. Now, we have to (re)define the semantics of the retrieval operation.

Retrieval in Temporal Databases

As a first objective, we have to guarantee that retrieval operations referring only to the actual data have to show the same behaviour in the temporal database and in the non-temporal database. As a second objective, we want to have powerful retrieval operations working on the historical data. These queries can be differentiated according to the kind of temporal qualification they use in the WHERE clause. Before we investigate these different kinds of temporal queries, we give a short overview of the syntax of the retrieval statements. In general, we use the syntax of the introduced MQL with some slight modifications.

T_SELECT projection_list **[temporal_projection]**
T_FROM object_definition
T_WHERE non_temporal_condition **temporal_selection**

temporal_selection ::= AT <time_point>|
{SOMETIMES | ALWAYS} DURING [time_point, time_point]

temporal_projection ::= CORRESPONDING |
AT time_point |
DURING [time_point, time_point]

In the following, we will discuss several kinds of temporal queries. We illustrate them by running some sample queries against our database as depicted in Figure 8. Since Figure 8 shows only a part of the database, most of the atoms have a validity interval from the first of January 1988 until now. The atoms connected by a bold edge are atoms constituting one TS. The TS reflect the changes to the database as shown in Figure 4. During the discussion of the query types, we neglect the temporal projection which we will discuss in a following step.

juncture query

This kind of query refers to the database state valid at a single point in time, like in the query "How did company *Cheese* look like at the 2nd of March 1989?". Perhaps, this is the most obvious way how the historical data can be accessed, i.e. we refer to a certain recent state of the database. The condition given in the WHERE clause is evaluated on the data valid at the given point in time†. Using the introduced syntax, we get the following temporal query:

† Thus, the company would also be retrieved, if it had changed its name after this date.

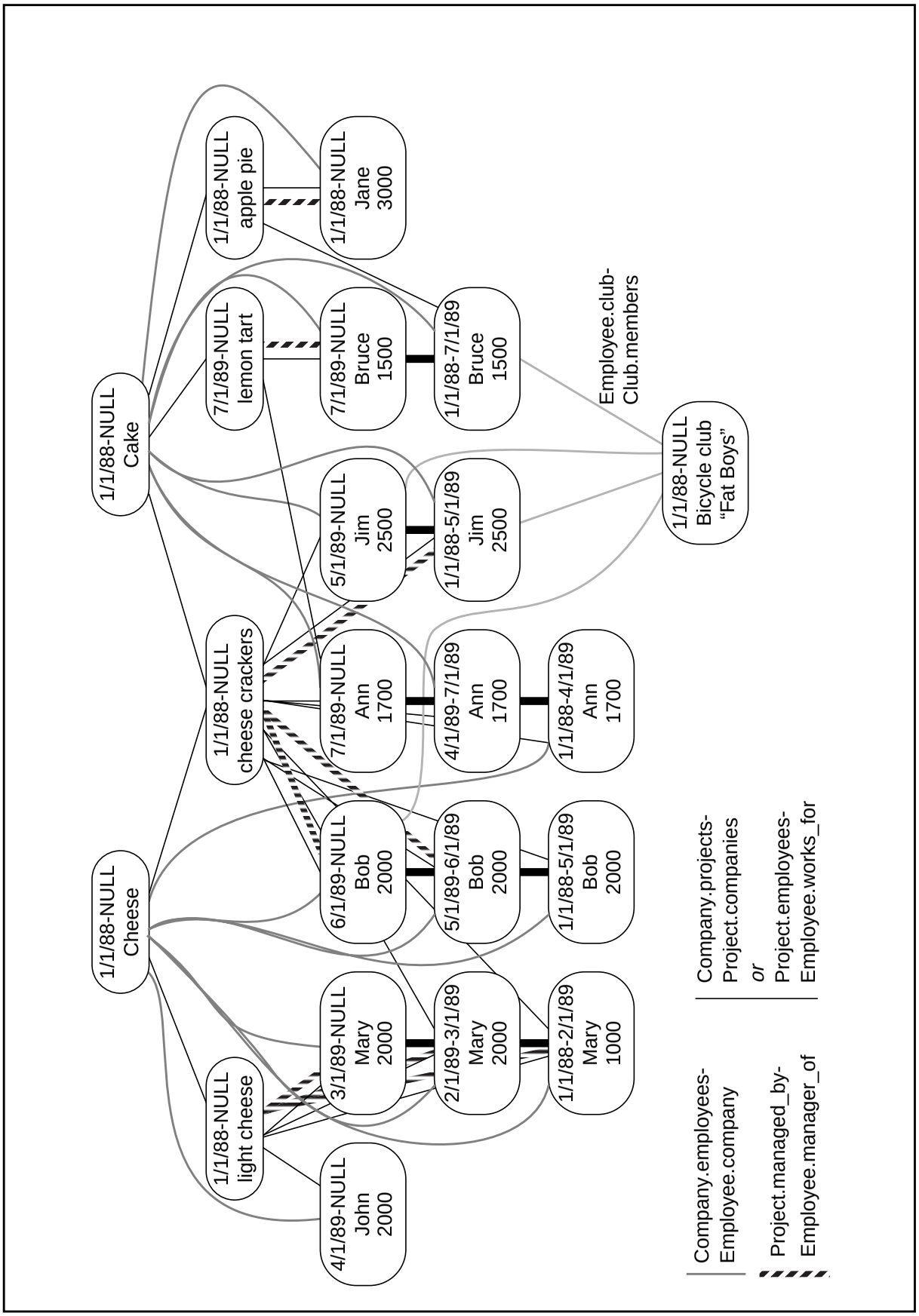

Figure 8: Sample database (representing all states of the data according to Figure 4)

(1) T_SELECT ALL
T_FROM Company-Project-(.employees-E(Employee), .managed_by-M(Employee))
T_WHERE company_name = “Cheese”
AT 3/2/1989

This query selects the values of all attributes of the companies (“Cheese”), the projects (“light cheese” and “cheese crackers”), their employees (“Mary”, “Bob”, “Jim” and “Ann”), and their managers (“Mary” and “Jim”) valid at the 2nd of March 1989.

As a special case, we can use the time function NOW which computes the actual date and time. Thus, the clause “AT NOW” retrieves the actual data from the database†.

existential interval query

The interval queries refer to an interval in time corresponding to one or several states of the data. In case of the existential interval query, the condition given in the WHERE clause has to hold at least for one point in time included in the given interval. For example, the query “Which clubs had members of company *Cheese* in the year 1989?” results in the following temporal query:

(2) T_SELECT club_name
T_FROM Club-Employee-Company
T_WHERE EXISTS Company: company_name = “Cheese”
SOMETIMES DURING [1/1/1989, 12/31/1989]

The query retrieves all clubs which have at least one member who worked for company “Cheese” at least for one point in time (within the given interval). In our sample database, we would get the bicycle club “Fat Boys”, because Bob joined the club at the 1st of June 1989.

universal interval query

In this case, the given condition has to hold for all states of the database valid in the given interval. Thus, the sample query (2)‡ would have an empty result, because Bob wasn’t member of the bicycle club in the first half of the year 1989. For another example, we could ask “For which projects did Mary work during the whole year 1989?”:

(3) T_SELECT project_name
T_FROM Project.employees-Employee
T_WHERE EXISTS Employee: employee_name = “Mary”
ALWAYS DURING [1/1/1989, 12/31/1989]

In our sample database, the query retrieves only project “light cheese”.

After having detailed the semantics of temporal selections, we have to take a closer look to the result of those queries.

Results of Temporal Queries

A query in the MAD model yields a set of molecules as its result. Analogously, a temporal query yields a set of molecule histories, which in turn are constituted of sets of molecules, each representing one

† We use this clause as default, if the query contains no explicit temporal selection.
‡ Of course, modified by changing SOMETIMES to ALWAYS.

state of the complex object valid for some interval of time (which we will specify later on). We call such a molecule together with the validity interval *time slice* (or shortly TSL). A time slice contains at most one state of each atom. As a consequence, the validity intervals of all atoms included in a time slice are equal. As an example, Figure 9 shows the time slices derived from a sample database of three temporal atoms. Obviously, each time slice equals to the result of a corresponding non-temporal query from a non-temporal database, asked at a point in time during the validity interval of the TSL.

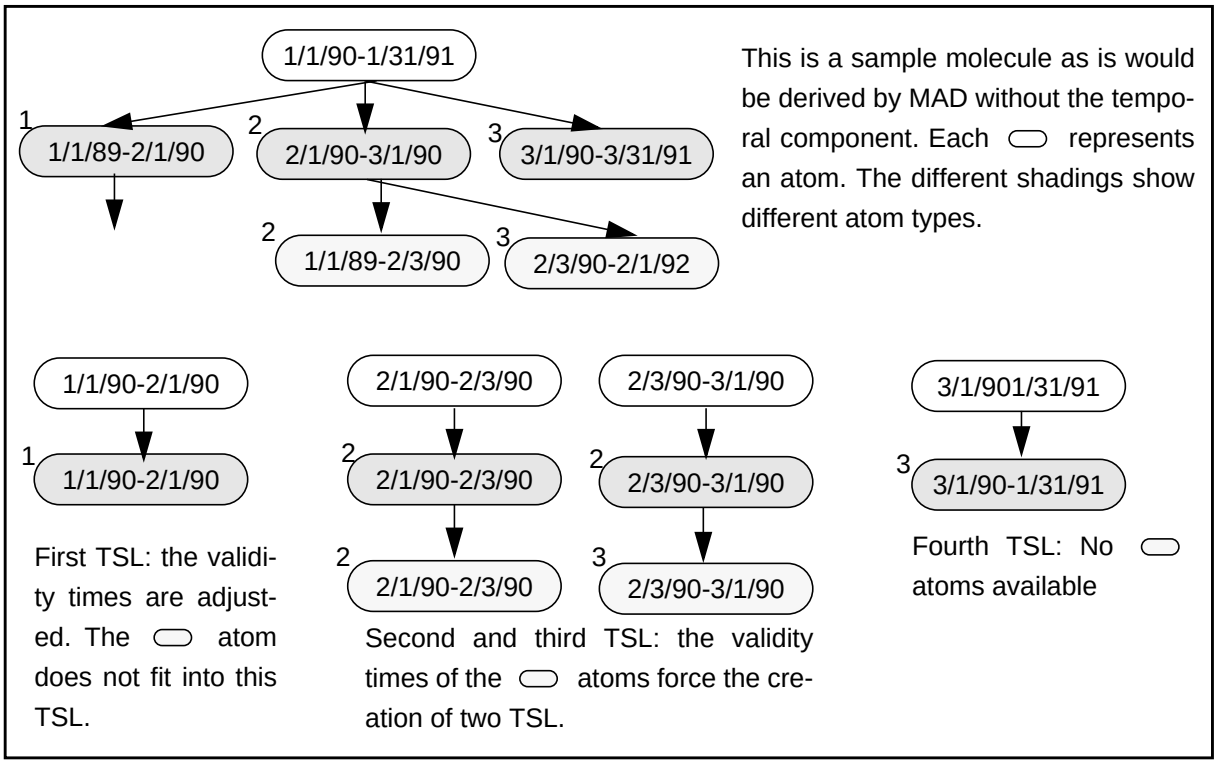

Figure 9: Example of a time slice (TSL) construction

In the case of the juncture query with the time selection “AT NOW” discussed above, each molecule history consists of exactly one time slice. Thus, the first requirement (queries referring to the actual state of the database generate the same result as in non-temporal MAD databases) is fulfilled.

However, the result of temporal queries becomes more complex, if we take a more general view: Selecting data at one special point in time (i.e. via a juncture query) does not necessarily mean that we want to get only the data valid at this special point in time. For example, we can ask for the actual salary of those employees who had worked for project “cheese crackers” at the 15th of February 1989. Thus, we have to apply a temporal projection which works like the qualified projection of the MAD model, i.e., the temporal projection selects those TSL from the result of a temporal query which are required by the user. For this purpose, we can apply an “AT” clause and a “DURING” clause in the *temporal projection*. The “CORRESPONDING” clause† leads to the selection of those TSL which were valid during the time specification given in the WHERE clause. The following example of a juncture query illustrates the effect of the different temporal projection clauses.

(4) T_SELECT employee_name, salary
T_FROM Project.employee-Employee

† We use this clause as default, if no explicit temporal projection is applied.

T_WHERE project_name = “cheese crackers”
AT 2/15/1989

- Using “CORRESPONDING” as temporal projection would retrieve “Mary 2000”, “Bob 2000”, “Ann 1700”, and “Jim 2500”.
- Using “AT 1/15/1989” as temporal projection would retrieve a salary of 1000 for Mary instead of 2000 as shown above.
- Using “DURING [1/15/1989, 3/15/1989]” as temporal projection would retrieve two TSL for Mary: “1000 for [1/15/1989, 31/1/1989]” and “2000 for [2/1/1989, 2/28/1989]”.

So far, we have introduced the TMAD model as a powerful data model for handling temporal complex objects. We have detailed various kinds of temporal queries and temporal answers. In the following chapter, we will discuss the implementation of the TMAD model as a layer on top of the MAD model.

5. Implementation of the Temporal model

In order to safe as much implementation work as possible, we decided to implement the temporal complex-object data model (TMAD) by an additional layer on top of the MAD model. Thereby, we want to delegate as much work as possible to the MAD model.

We describe the mapping process beginning with the building blocks of the TMAD model, the temporal atoms. As already mentioned, there have to be the additional attributes *valid_from* and *valid_until* as well as the *previous* and the *next* REF_TO attribute. Thus, each user-defined temporal atom type is transformed to a MAD atom by adding these four attributes and omitting the upper bounds of the cardinality restrictions. Manipulation of temporal atoms has already been described. Only the handling of retrieval statements has still to be discussed.

We transform each TMAD query to a MAD query which defines one molecule for each molecule history.

juncture query

Juncture queries are easy to transform: we only have to find one time slice by evaluating the *valid_from* and *valid_until* attribute (Example 1)

T_SELECT ALL
T_FROM Club-Employee
T_WHERE employee_name = “Mary” AT 3/1/1989

is transformed to

SELECT ALL
FROM Club-Employee
WHERE employee_name = “Mary” AND
Club.valid_from ≤ 3/1/1989 AND
((Club.valid_until > 3/1/1989) OR IS_NULL (Club.valid_until))

Example 1: Transformation of a juncture query

It is sufficient to ask only for the *valid_from* and *valid_until* attributes of the root atom type, because it contains all references which were relevant in the interval formed by *valid_from* and *valid_until*. The *valid_until* attribute may have a NULL value, if the atom is representing the actual state.

universal interval query

In this case, we have to retrieve a set of temporal molecules, each of them representing a sequence of different states of the molecule valid during one interval in time. For this purpose, we first retrieve the MAD molecule which represents the latest state of the temporal molecule as referenced in the query. We then append all states until we reach the lower bound of the time interval by using the MAD models recursion facility. Hence, we use the *next* and *previous* attributes of the root atom type of the molecule. Example 2 illustrates our approach.

```
T_SELECT    ALL
T_FROM      Company-Employee
T_WHERE     company_name = “Cake”
            ALWAYS DURING [2/1/1989, 5/1/1989]
```

is transformed to

```
SELECT      ALL
FROM        (Company-Employee) RECURSIVE Company.previous-Company
            UNTIL Company.valid_until < 2/1/1989
WHERE       Company(0).valid_from ≤ 5/1/1989 AND
            (Company(0).valid_until > 5/1/1989 OR
            IS_NULL (Company(0).valid_until)) AND
            Company(LAST).valid_from ≤ 2/1/1989 AND
            FOR ALL Company(ALLREC):
            Company(ALLREC).company_name = “Cake”
```

Example 2: Transformation of a universal interval query.

The UNTIL clause stops the construction of the recursive molecule when the interval is overstepped. The condition “Company(Last).valid_from ≤ 2/1/1989” guarantees that the complex object existed during the whole interval.

Here, the condition is included into the MQL query because it only restricts the root atom type. In general, the condition cannot be evaluated by MAD because it is time specific. For example, consider the condition “SUM(salary)<8000” added to the above query. The molecule constructed by the MAD query contains two atoms of “Ann”. Thus, regarding the molecule, the condition would be violated, although it holds for each time slice.

existential interval query

In the case of an existential interval query the condition “Company(LAST).valid_from ≤ 2/1/1989” is omitted and the “FOR ALL” is replaced by “EXISTS”. The remarks concerning the evaluation of the condition by MAD hold also for this kind of queries.

construction of time slices

The result of such queries is a set of recursive molecules each belonging to one complex object. Each recursion level of such a molecule corresponds to one state of the complex object, starting with the most recent state within the specified interval. These molecules have now to be transformed into a set of time slices for each complex object. This is done by the temporal layer. First, the validity intervals of the time slices have to be computed. All *valid_from* (or *valid_until*) attributes in the molecule, which do not exceed

the interval specified by the query are sorted. In this sorted list, each two adjacent values form the validity intervals of a time slice†. For each time slice, the corresponding root atom is determined. The construction mechanism of the time intervals for each TSL guarantees that each time slice contains at most one state of each temporal atom. From the root atom, the links are followed, thereby discarding all atoms which do not overlap the time interval of the TSL. The validity time of all atoms of a TSL is set to that of the TSL itself. As soon as the TSL are constructed, the conditions which could not be transferred to MAD must be tested. In the case of an existential interval query the condition has to hold for at least one of the TSL, in the case of a universal interval query for all TSL.

Until now, we implicitly required the projected time to be a sub-interval of the time of the “time selection clause”.

This makes the mapping process easier, but is not a necessary restriction. We can use the methods discussed in [KRS90] to overcome it.

Alternatively, the temporal layer may perform its work in two steps in the case where the projected time is not a sub-interval of the time of the time selection clause. In a first step, the qualifying molecules are computed as described above and the identifiers of the root atoms are memorized. Then, a second query with the only condition “Recursive molecule contains one of the identifiers memorized in step 1” is used to retrieve the final result.

As mentioned above, during the construction of a time slice, atoms which are not valid during the validity interval of the time slice, are discarded. Hence, atoms which are outside of the interval specified in the temporal selection clause are always discarded by the temporal layer. Obviously, the MAD queries as described above deliver a couple of such atoms, as the following example illustrates:

temporal query:

```
T_SELECT    ALL
T_FROM      Company-Employee
T_WHERE     company_name = “Cheese” AT 5/1/1989
```

MAD query:

```
SELECT      ALL
FROM        Company-Employee
WHERE       Company.Company_Name = “Cheese” AND
            Company.valid_from ≤ 5/1/1989 AND
            (Company.valid_until > 5/1/1989 OR IS_NULL (Company.valid_until))
```

Example 3: Sample juncture query transformation

In our sample database, there is only one *Company* atom representing *Cheese*. Hence, this atom as well as all referenced employees including Ann are retrieved. However, Ann does not belong to *Cheese* at 5/1/1989. The corresponding atom could be discarded by the additional temporal layer. We prefer discarding it already in the MAD query by using the concept of qualified projection for each atom type involved. Thus the optimized transformation result is:

† The first and the last value of the list form an interval with the corresponding bounds of the interval specified in the query.

```
SELECT    Company,   SELECT    ALL
                     FROM      Employee
                     WHERE     Employee.valid_from ≤ 5/1/1989 AND
                               (Employee.valid_until > 5/1/1989 OR
                               IS_NULL(Employee.valid_until))
FROM      Company-Employee
WHERE     Company.Company_Name = “Cheese” AND
          Company.valid_from ≤ 5/1/1989 AND
          (Company.valid_until > 5/1/1989 OR IS_NULL (Company.valid_until))
```

Analogously, for interval queries, all atom types except the root atom type undergo the qualified projection.

6. Conclusion and Outlook

Comparison to the TMQL model

In [KRS90] we presented a temporal extension to the relational model. There, we added only one “valid” attribute to the relations (having the semantics of *valid_from*). *valid_until* was represented in the next atom of a time sequence. This prevents “holes” in the time sequence already by the way of modelling. Furthermore, the insertion of a new state of an atom did not force an explicit change of an attribute (*valid_until*) of the previous atom. However, retrieval queries become more complex. In order to stress the complex-object aspects of our approach presented in this paper, we have chosen the more intuitive approach of both attributes. Nevertheless, there is no reason why the approach of [KRS90] cannot be applied here. Also, the storage saving techniques presented there are applicable.

Conclusion

We have shown that it is possible to extend a complex object data model by a temporal dimension without a huge amount of overhead. We could use all facilities of the underlying complex-object data model, and had only to restructure the results of queries to this model. One temporal query corresponds to only one non-temporal query or to two queries if the less sophisticated method of handling projections outside the selection interval (cf. section 5) is chosen.

The basic idea of our approach is not to represent a change to a complex object by storing the new state of the whole complex object, but rather by representing only the new state of the building block which had changed. This is a prerequisite for incorporating dynamic complex object definitions at query time into the temporal complex-object data model. Furthermore, this approach does not require a huge storage overhead, even without additional compression techniques like the ones presented in [KRS90].

Notice that the concept of a unique identifier for each temporal atom is not automatically supported by our approach. We do not see the need for such an identifier (because it can neither be used for references nor is it necessary to group the various states of a temporal atom by such an identifier, because they are already linked to one another by the REF_TO attributes *past* and *future*). Nevertheless, a unique identifier could be computed by the temporal layer at the creation time of a temporal atom and might be added to the atoms’s schema definition. For the underlying MAD database, it will appear as a normal attribute.

As a further work, we will consider whether there are any other classes of temporal queries which make sense in the complex-object context (perhaps the “coincidence” queries of [KRS90]). Furthermore, we have to investigate in which case conditions which do not only restrict the root atom type may be transferred to the MAD query. We will also have to study the impact of clustering mechanisms on the performance of our temporal database system. Obviously, we cannot cluster all states of a molecule physically. We have to investigate whether there are any tailored access paths for our approach.

We claim that the temporal extension presented in the context of the MAD model is also possible for similar complex-object data models. As a result of our work, we conclude that implementing a temporal complex-object data model quite efficiently is not much harder than implementing a non-temporal complex-object data model.

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.

AS86: Ahn, I., Snodgrass, R.: Temporal Databases, IEEE COMPUTER, Vol. 19, No. 9, 1986, pp. 35-42.

BK85 Batory, D.S., Kim, W.: Modelling Concepts for VLSI CAD Objects, in: ACM TODS, Vol. 10, No. 3, 1985, pp. 322-346.

DL88 Dittrich, K.R., Lorie, R.A.: Version Support for Engineering Database Systems, in: IEEE Transactions on Software Engineering, Vol. TOSE-14 (1988), No. 4, pp. 429-437.

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.

HMMS87: Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. Int. Conf. on Very Large Data Bases VLDB’87, Brighton, 1987, pp. 433-442.

Kä91: Käfer, W.: A Framework for Version-based Cooperation Control, Proc. of the 2nd Int. Conf. on Database Systems for Advanced Applications (DASFAA), Tokyo, Japan, 1991.

KRS90: Käfer, W., Ritter, N., Schöning, H.: Support for Temporal Data by Complex Objects, in: Proc. Int. Conf. on Very Large Data Bases VLDB’90, Brisbane, Australia, 1990, pp. 24-35.

KS91: Käfer, W., Schöning, H.: Mapping a Version Model to a Complex-Object Data Model, Research Report No. 22/91, University Kaiserslautern, 1991.

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.

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.

Schö90 Schöning, H.: Preserving Consistency in Nested Transactions, in: Proc. 23rd Annual Hawaii Conf. on System Sciences, HICSS23, Vol. II, IEEE, 1990, pp. 472-480.

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.

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