in : Proc. SIGMOD 1992, San Diego, pp. 266-275.

Realizing a Temporal Complex-Object Data Model

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. Furthermore, the strong demand for complex-object support is known as an inherent fact in design applications, and also emerges for advanced "conventional" applications. Thus, new advanced database management systems should include both features, i.e. should support *temporal complex-objects*. In this paper, we present such a temporal complex-object data model. The central notion of our temporal complex-object data model is a *time slice*, representing one state of a complex object. We explain the mapping of time slices onto the complex objects supported by the MAD model (which we use for an example of a *non-temporal* complex-object data model) as well as the transformation process of operations on temporal complex-objects into MAD model operations. Thereby, the basic properties of the MAD model are a prerequisite for our approach. For example, time slices can only be directly stored, if non-disjunct (i.e. overlapping) complex objects are easily handled in the underlying complex-object data model.

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. [Da88, Ga88, GY88, Sn86, SS88, Ta86]. 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.

The relational model has been found useful in commercial applications, but there are application areas where more powerful data models are required, for example CAD, expert systems or geographic applications. For these areas, complex-object data models have been developed which map each entity of the real world to only one object in the databases, whereas the relational model in general maps it to many tuples of different relations due to the normalization required by this model. For example, the entity type "company" (which consists of employees, projects, plants, etc.) is mapped to a set of relations when using the relation model, whereas a complex-object data model maps is to a single (complex) object type. We claim that complex-object data models are useful 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.

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 of a complex-object data model. In particular, changes to the building blocks of a complex object (for example, raise of the salary of an employee of the company) and to the relationships among them (for example, a change in the project assignment of an employee) must be captured in a natural way.

Modelling complex objects with the relational model has been found very cumbersome, mainly because of the great number of joins to be performed. Looking at temporal extensions of the relational model, we can recognize that the temporal evolution of objects (tuples) is captured quite well. However, changes in the relationships among objects are not easily handled, because in general two tuples may be joined only when they are temporally coincident. For this purpose, various forms of a "temporal join" have been introduced [GS90, SG89]. All of them are very complex operations and therefore even more expensive than the usual join operation in relational database systems. We will see that modelling complex objects with the help of a *temporal complex-object* data model instead of the temporal relational model relieves us from the burden of a temporal join facility.

We did not choose an example from a design environment or another advanced application area to illustrate our approach, because we wanted to avoid the introduction of these sophisticated environments here. Nevertheless, the reader should observe the correspondence of the temporal development of the complex objects in our example and the temporal development of complex objects in advanced applications. Design environments (and also almost all other application 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. Versioning tends to be non-linear, i.e., contain branches and alternatives. These aspects of temporal data are covered in [KS92] 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 the facts in the history of the complex objects which will be of any interest in the future are not known in advance. Furthermore, the semantics of a version (i.e. a distinguished (intermediate) result or unit of cooperation) do not apply to the history aspect. Time in this context is linear.

In this paper, we deal with the latter aspect (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 *as the data model to be extended* by a temporal dimension as well *as to implement this extension by mapping it onto the MAD model* (i.e. by performing transformations as depicted in Figure 1). We call the temporally extended MAD model *TMAD* model, for short. 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.

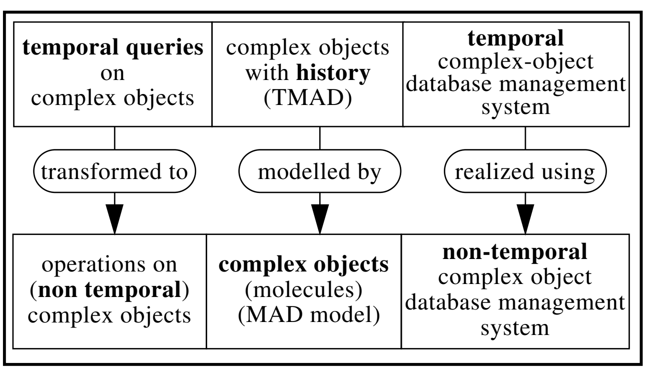


Figure 1: The mapping of the temporal data model to the complex-object data model

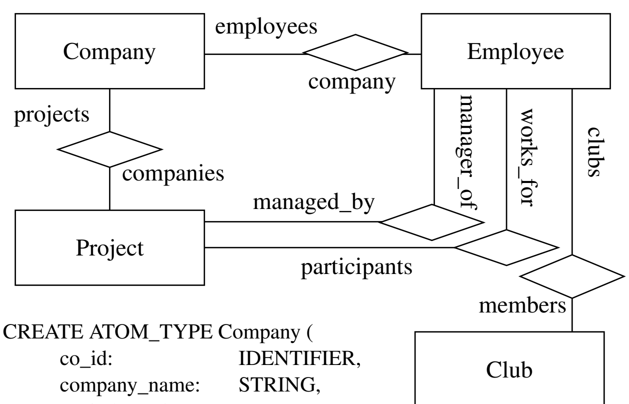
The following section gives a brief description of the MAD model and introduces our running example. The query facilities of the model are exemplified on a small sample database. Section three deals with the temporal extensions of the MAD model thereby defining the TMAD model, which is again illustrated on our sample database. The fourth section discusses the implementation of the TMAD model by means of the MAD model. We close with a short summary and some conclusions.

2. The Molecule-Atom Data 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 open only to the employees of the companies. 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.

E/R diagram of the sample mini world




```
CREATE ATOM_TYPE Company (
    co_id:            IDENTIFIER,
    company_name:     STRING,
    capital_stock:    INTEGER,
    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.participants) (1,*),
    manager_of:       REF_TO (Project.managed_by) (0,1),
    clubs:            REF_TO (Club.members) (0,*));

CREATE ATOM_TYPE Project (
    proj_id:          IDENTIFIER,
    project_name:     STRING,
    budget:           INTEGER,
    participants:     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.clubs)(2,*));
```

Figure 2: Database schema of the sample database

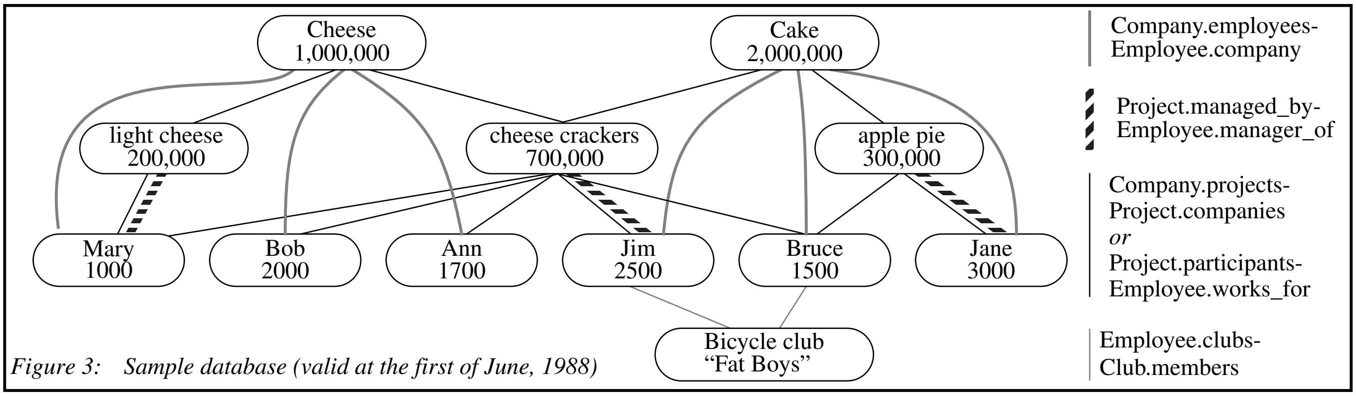


Figure 3: Sample database (valid at the first of June, 1988)

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[1]. A MAD model schema definition 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.clubs*. Each instance of an atom type (called atom) has a unique system-controlled identifier. The 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. 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 two members, and an employee works exactly for one company according to the schema depicted in Figure 2. A sample database is shown in Figure 3. The "bubbles" represent atoms, whereas the lines represent the links between them.

Using the MAD model's query language MQL, one can retrieve the complex object "Cheese" (company including projects and their employees) by the following query: (Query 2.1)

```
SELECT   ALL
FROM     Company.projects-Project.participants-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 *participants* REF_TO attribute.

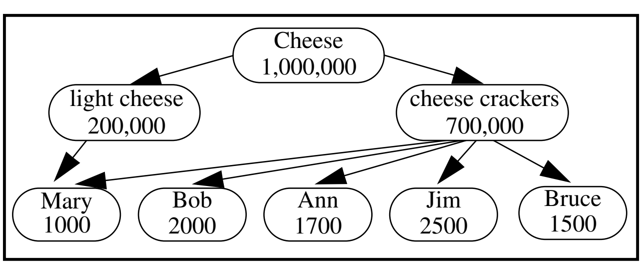


Figure 4: Result of Query 2.1

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 (*molecule type definition*), and the WHERE clause is used to specify which molecules of that type qualify. The result of a query is a set of molecules, the type of which is dynamically defined at query time.

The result of this query applied to the database of Figure 3 is shown in Figure 4. The query only includes information about the employees, but not about the managers of the projects. We can change it to obtain this information by following two paths (i.e., links of two different types) to *Employee*. To be able to distinguish the two occurrences of *Employee* in this query, we assign so-called role names (P and M) to them. The following query also shows the use of the projection clause.

```
SELECT Company(capital_stock), Project (Budget),
       P (salary, employee_name), M (employee_name)
FROM   Company.projects-Project-(.participants-P (Employee),
                                 .managed_by-M (Employee))
WHERE  company_name = “Cheese”
```

The result of this query contains one molecule consisting of an atom representing the company's capital stock, one atom for each project (showing the budget of the project), atoms for each employee as depicted in Figure 4, and additionally the atoms indicating that Mary and Jim are managers. The atoms are interconnected by appropriate links. Regarding Figure 4, we notice that query 2.1 includes the projects "cheese crackers" and "light cheese" and the employees "Mary", "Bob", "Ann", "Jim", and "Bruce" 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.participants-Employee
WHERE  company_name = “Cheese”[2]
```

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).

If we want to get all projects of a company, but only the company's own employees, we can specify this by using the network semantics of the MAD model: if there is more than one reference attribute pointing to an atom type, atoms of this type are included only if they are reachable via all the references attributes:

1.with binary attribute-free relationships of type 1:1, 1:n, or n:m.

2.The name of a REF_TO attribute may be omitted in the molecule type definition, if there is only one such attribute connecting the two atom types.

SELECT ALL
FROM Company - (.projects - Project.participants-Employees,
.employees-Employees)
WHERE company_name = “Cheese”

Furthermore, the MAD model can handle recursively structured complex objects. Suppose, for example, one wants to retrieve all employees who directly or indirectly come into contact to “Mary”. Two employees come into contact if they work in the same project. We can retrieve this information by the following query:

SELECT ALL
FROM Employee.works_for - Project
RECURSIVE Project.participants - Employee
WHERE Employee(FIRST).employee_name= “Mary”

The keyword RECURSIVE forces the construction of recursive complex objects (transitive closure). An additional UNTIL clause may be used for the premature termination of the recursion.

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 5. These changes may be applied to the database using the MQL statement UPDATE (replacing the old values by the new ones). Unfortunately, the MAD model discussed so far is capable only of showing the latest state of the database. Figure 6 shows the database state after all these changes have been performed.

2/1/1989:	Mary’s salary is raised to 2000
	The capital stock of “Cake” is changed to 2,500,000
3/1/1989:	Mary quits from project “cheese crackers”
4/1/1989:	Ann quits company “Cheese” and joins “Cake”
	John joins company “Cheese”, project “light cheese”
5/1/1989:	Bob becomes Manager of “cheese crackers”
6/1/1989:	Bob becomes member of bicycle club “Fat Boys”
	The budget of “light cheese” is raised to 400,000
7/1/1989:	A new project “lemon tart” is created by “Cake”
	Bruce changes to “lemon tart”, becomes manager
	Annworks for “lemon tart” and “cheese crackers”

Figure 5: Changes applied to the sample database during 1989

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?”
- “Which projects had a budget of 400,000 or more when one of their companies had a capital stock of 1,000,000 or less?”

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.

3. The Temporal Complex-Object Data Model

In this section, we will discuss the temporal data model TMAD which we want to implement. As mentioned before, it is an extension of the MAD model, and therefore, we will base the introduction of the temporal model on the MAD model. We start we the objects known by TMAD, which are molecule histories. We then postulate the retrieval facilities needed in TMAD, and end with a discussion of manipulation operations. We do not address schema definition aspects, because they can be derived from the MAD model in a straightforward way.

Objects of TMAD

The objects dealt with by the MAD model are molecules: an MQL query yields a set of molecules as its result, and manipulation statements manipulate (sets of) molecules. Hence, the objects of a temporally extended MAD model are temporally extended molecules (*molecule histories*), and a temporal query yields a set of molecule histories as result. A molecule history describes the evolution of a molecule in a certain interval of time (*validity interval*). It consists of all states the molecule has had during the interval. Each state is a molecule enhanced by a validity interval specifying when the molecule had this state. We call such an enhanced molecule *time slice* (TSL). Obviously, each time slice equals (apart from the validity interval) 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.

Retrieval in Temporal Databases

In this section, we identify the retrieval facilities required for TMAD. As a first objective, we have to guarantee that retrieval operations referring only to the actual data 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. We precede all operations of our temporal model with T_ in order to distinguish them from the MAD operations. 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 (which assumes that the mini world exists since the first of January 1988). The representation of the database reflects the fact that atoms have developed in time (cf. Figure 5). The bubbles connected by a bold edge are states of one atom, each of them valid during the corresponding validity interval. During the following discussion of temporal query types we will neglect the temporal projection which we will detail 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 "Which companies had a capital stock of 1,000,000 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:

```
(1) T_SELECT  ALL
    T_FROM    Company-Project-( .participants-E(Employee),
                                .managed_by-M(Employee))
    T_WHERE   capital_stock = 1000000 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.

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)[1] 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.participants-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".

Coincidence Query

This kind of query checks whether two conditions hold at the same time. Whereas in the temporally extended relational models complex join operations have to be defined to express this kind of query, if it spans more than one relation, the complex object approach allows for a natural formulation of the query. Thus the query "Which projects had a budget of 400,000 or more when one of their companies had a capital stock of 1,000,000 or less?" delivers project "light cheese" when applied to our sample database:

```
(4) T_SELECT  project_name
    T_FROM    Project - Company
    T_WHERE   Project.budget >= 400000 AND
              EXISTS Company:
              Company.capital_stock <= 1000000
              SOMETIMES DURING [1/1/1988, NOW]
```

Temporal Projection

Until now, we assumed that the result of each query reflects the interval mentioned in the WHERE clause. See, for example, query (1) where we said that the result was the state of the company at March 2nd, 1989. If we use the time selection "AT NOW", each molecule history consists of exactly one time slice showing the actual state of the database. 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 projec-

1.Of course, modified by changing SOMETIMES to ALWAYS.

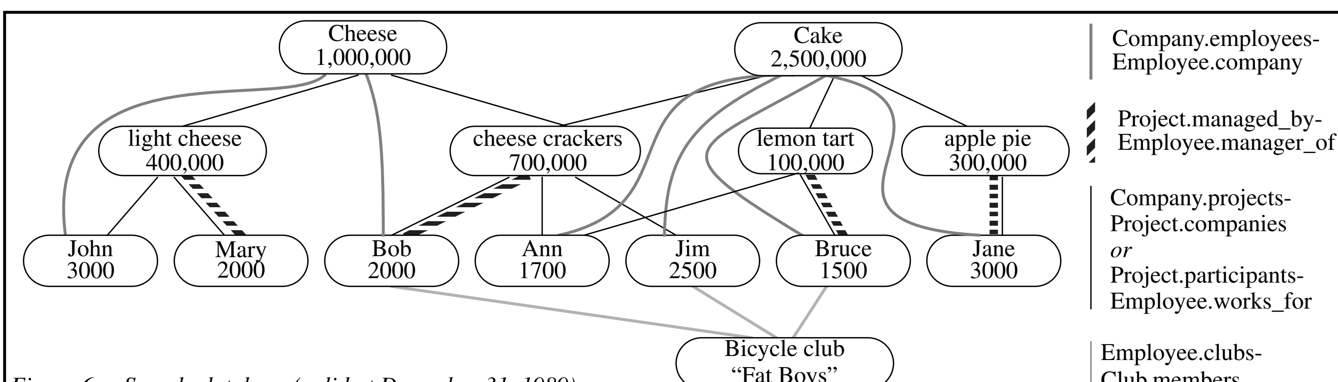


Figure 6: Sample database (valid at December 31, 1989)

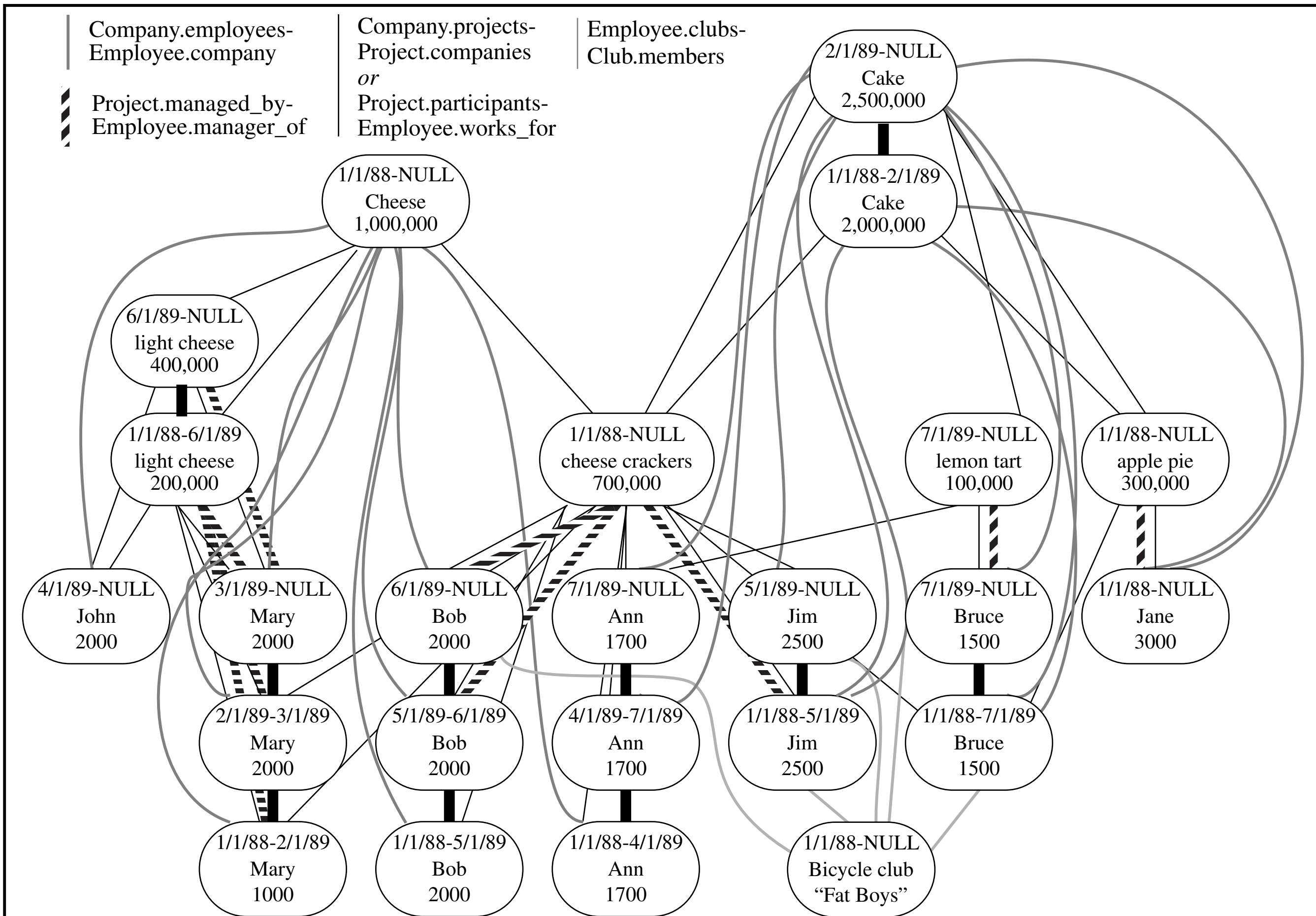


Figure 7: Sample database (representing all states of the data according to Figure 5)

tion 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[1] 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.

```
(5)  T_SELECT   employee_name, salary
     T_FROM     Project.employee-Employee
     T_WHERE    project_name = "cheese crackers"
                AT NOW
```

- 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 "Mary 1000", "Bob 2000", "Ann 1700", "Jim 2500", and "Bruce 1500".
- Using "DURING [1/15/1989, 3/15/1989]" as temporal projection would retrieve two TSL for Mary: "1000 for [1/15/1989, 1/31/1989]" and "2000 for [2/1/1989, 2/28/1989]".

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

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 inserts copies of the latest state of all modified atoms into the database without deletion of the old states. Each state is marked with a corresponding validity interval. The user has to specify the validity time in the T_UPDATE statement (in the form VALID_FROM t_1 (and optionally VALID_UNTIL t_2), because this time reflects when the associated fact will be valid in the mini-world. The validity time must not be confused with the transaction time, which represents the time, when a fact is inserted in the database. The transaction time is automatically recorded by our system.

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 by a layer on top of the MAD model.

4. Implementation of the Temporal Model

The TMAD model is implemented in large parts by simple transformation processes. For example, the schema definition of TMAD atom types is simply extended to get a suitable MAD model schema definition. Similarly, the T_UPDATE operation is realized by query transformation. Only the T_SELECT operation needs some more efforts to be implemented.

Transformation of a TMAD Schema into a MAD Model Schema

The first problem we have to solve when implementing TMAD on top of the MAD model is the representation of data. While MAD model databases contain atoms, we have to store temporal atoms now, i.e. the collection of all different states of one (logical) atom together with their validity intervals. We record each state of a temporal atom by storing the attribute values of the state together with the validity interval as one MAD atom. For this purpose, we add the attributes *valid_from* and *valid_until* to each atom type definition. The transaction time is stored in the attribute *transaction_time*. Furthermore, all states of an atom are connected to one another by the attributes *past* and *future*. shows the resulting MAD model schema definition for the TMAD atom type Company.

```
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),
    transaction_time:   TIME(SECOND),
    past:               REF_TO (Company.future) (0,1),
    future:             REF_TO (Company.past) (0,1));
```

Figure 8: Schema extension of an atom type in order to capture the historical data

Implementation of T_UPDATE

When performing a T_UPDATE operation, we have to consider two points, because the atom representing the old state and the one representing the new (modified) state logically represent one temporal atom. Firstly, we have to mark each atom with its validity interval. Secondly, we have to combine the atoms which build the temporal atom.

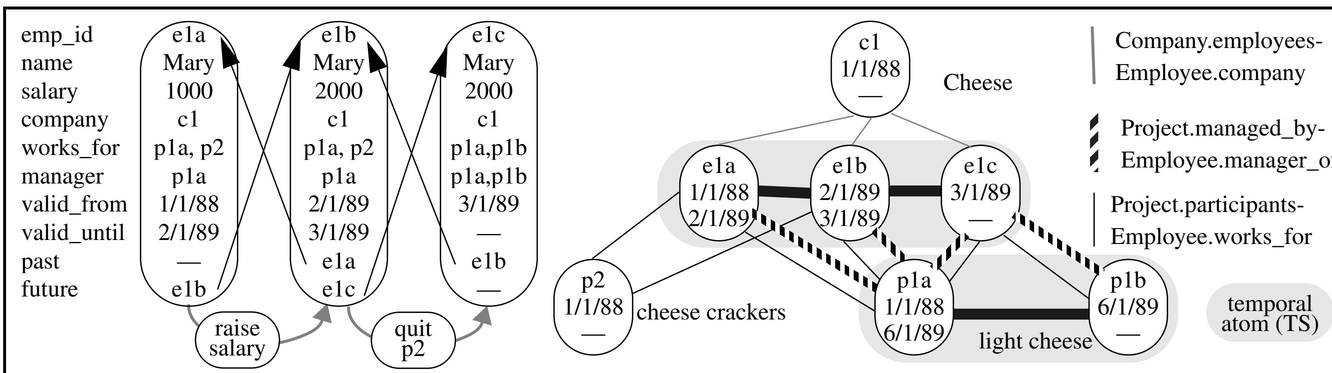
We accomplish the first task by assigning appropriate values to the attribute pair *valid_from* and *valid_until*[1] of the atom. The T_UPDATE operation has to ensure that the *valid_from* attribute of the new atom contains the same value as the *valid_until* attribute of the previous atom in order to avoid “holes” in the history of the atom.

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 the REF_TO attributes *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, SK86]. Thus, a temporal atom is stored as a TS.

As an example, a shows the TS representing Mary’s history according to our mini-world (transaction time has been left out). 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, b reflects the fct that Mary’s history is only one part of the database and therefore the atoms are connected with other atoms of the database which may belong to the same or to different TS (the whole database is shown in). Raising Mary’s salary to 2000 at the 1st of February 1989 forces the insertion of a new atom into the database as described in the above discussion of 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[2]. 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 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. The effect of the referential integrity maintenance mechanism described above is shown in , where the attribute *works_for* of object e1c is expanded by p1b due to the change of the budget of project light cheese.

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

2. 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.



a) The TS of Mary; p1: light cheese, p2: cheese cracker, c1: Cheese

b) The TS in the context of the database

Figure 9: History of Mary

As a consequence of our approach, cardinality restrictions have to be reconsidered. The upper bounds may not longer hold, if an atom references more than one atom of the same TS. For example, in the database shown in , 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 TMAD schema into a MAD schema, and check them in the additional temporal layer which we will introduce below.

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*), one attribute for the transaction time, and the implementation of the T_UPDATE operation by simple query transformation is sufficient for storing the historical data along with the actual data in the database.

Implementation of T_SELECT

The retrieval procedure is much more complicated. It bases on the notion of time slices, as introduced before. A TSL is defined as the molecule's state valid within a specific time interval such that it contains only one state of each atom of the molecule. Unfortunately, we cannot force MQL to produce such molecules. However, we can retrieve the set of all relevant data for each molecule history as a single molecule. As a consequence, we transform a TMAD query such that it delivers sets of (possibly recursive) molecules, one set for each TSL of the result. These molecules are then transformed into TSLs by an additional layer.

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 10)

```
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.employee_name = "Mary" AND
         Club.valid_from ≤ 3/1/1989 AND
         ((Club.valid_until > 3/1/1989) OR
         IS_NULL (Club.valid_until))
```

Example 10: 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. Note that there may be atoms of type Employee in the resulting molecule which are not valid in the specified interval. They are filtered out in a subsequent step (see below).

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 recent states until we reach the lower bound of the time interval by using the MAD models recursion facility. Hence, we use the *past* attribute of the root atom type of the molecule. Example 11 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)
         REC_PATH Company.past-Company UNTIL
          Company(CURRENT).valid_until < 2/1/1989
WHERE    Company(FIRST).valid_from ≤ 5/1/1989 AND
         (Company(FIRST).valid_until > 5/1/1989 OR
          IS_NULL (Company(FIRST).valid_until)) AND
         Company(LAST).valid_from ≤ 2/1/1989 AND
         FOR ALL Company(ALL_REC):
         Company(ALL_REC).company_name = "Cake"
```

Example 11: Transformation of a universal interval query.

The UNTIL clause stops the construction of the recursive molecule when the given interval is overstepped. The condition "Company(LAST).valid_from ≤ 2/1/1989" guarantees that the molecule 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 the MQL queries presented above is a set of recursive molecules each containing all relevant data of exactly one molecule history (and possibly some irrelevant data). Each recursion level of such a molecule corresponds to one state of the molecule's root atom, starting with the most recent state within the specified interval. Each of these molecules must now be transformed into a set of time slices giving a molecule history. This is done by the additional 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[1]. 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

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

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 12: 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:

```
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.

5. Conclusion and Outlook

Comparison to the TSQL model

In [KRS90] we presented TSQL as 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

In this paper, we have presented a temporal data model which fulfils the requirements listed in [WD92], i.e., support for various temporal operators, manipulation of temporal complex objects, modal operators, etc. Here, we have dealt only with a linear notion of time. The non-linearity found in versioning models can be implemented using the same approach, as we have shown in [KS92]. We believe, however, that the operations and semantics to be provided for versioning significantly differ from those of historical models. Hence, we define separate data models for these two areas. [WD92] burdens the database designer with the definition of the temporal data model, thereby achieving a uniform treatment of complex objects in linear and non-linear environments. We feel that the definition of a temporal data model is not trivial and therefore should be provided to the database designer.

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 4) 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 a more general version of the "coincidence" queries). 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. Other proposals like [WD92] lack hints for an efficient implementation.

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.

Da88 Date, C.-J.: A Proposal for Adding Date and Time Support to SQL, ACM SIGMOD RECORD, Vol. 17, No. 2, June 1988, pp. 53-76.

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.

GS90 Gunadhi, H., Segev, A.: A Framework for Query Optimization in Temporal Databases, in: Michalewicz (ed.): Statistical and Scientific Database Management, 5th Int. Conf. on SSDBM, Charlotte, N.C., USA, Apr. 1990, pp. 131-147.

GY88 Gadia, S.-K., Yeung, C.-S.: A Generalized Model for a Relational Temporal Database, Proc. ACM SIGMOD Int. Conf. on Management of Data, Chicago, Illinois, 1988, pp. 251-259.

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.

KS92 Käfer, W., Schöning, H.: Mapping a Version Model to a Complex-Object Data Model, in: Proc. of the 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992, pp. 348-357.

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, The Netherlands, 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.

SG89 Segev, A., Gunadhi, H.: Event-Join Optimization in Temporal Relational Databases, Proc. 12th Int. Conf. on VLDB, Kyoto, Japan, 1986, pp. 79-88.

SK86 Shoshani, A., Kawagoe, K.: Temporal Data Management, Proc. 15th Int. Conf. on VLDB, Amsterdam, The Netherlands, 1989, pp. 205-217.

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

SS88 Stam, R.-B., Snodgrass, R.: A Bibliography on Temporal Databases, Database Engineering, Vol. 7, 1988, pp. 231-239.

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

WD92 Wu, G.T.J.; Dayal, U.: A Uniform Model for Temporal Object-Oriented Databases, in: Proc. 8th Int. Conf. on Data Engineering, Tempe, AZ, 1992, pp. 584-593.